Broadview®
www.broadview.com.cn

Packt›

Native Docker Clustering with Swarm

Swarm容器编排与Docker原生集群

用Swarm部署、配置和运行Docker容器集群

[俄] Fabrizio Soppelsa
[泰] Chanwit Kaewkasi
著
崔婧雯　钟最龙　译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Native Docker Clustering with Swarm

Swarm容器编排与Docker原生集群

[俄] Fabrizio Soppelsa
[泰] Chanwit Kaewkasi
著
崔婧雯　钟最龙　译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Docker Swarm 作为 Docker 集群原生的容器编排解决方案，是 Docker 生态系统中的关键组件之一。本书涵盖了 Swarm 中的发现、调度、高可用、安全和平台伸缩性等重要主题，能帮助你了解 Swarm 如何组建包含 4700 个节点的集群，并掌握 Swarm 的使用与管理，以及如何使用实现大规模应用的可伸缩。

本书适合企业架构、开发、运维等各岗位从业者阅读，同样适合广大想了解当前主流 CaaS 架构内在运行原理与真实场景实践的普通学习者。

版权贸易合同登记号　图字：01-2017-2760

图书在版编目（CIP）数据

Swarm 容器编排与 Docker 原生集群 /（俄罗斯）法布里齐奥 •索贝尔萨（Fabrizio Soppelsa），（泰）占伟 • 瓦卡斯（Chanwit Kaewkasi）著；崔婧雯，钟最龙译. —北京：电子工业出版社，2017.7

书名原文：Native Docker Clustering with Swarm

ISBN 978-7-121-31792-7

Ⅰ. ①S… Ⅱ. ①法… ②占… ③崔… ④钟… Ⅲ. ①Linux 操作系统－程序设计 Ⅳ. ①TP316.85

中国版本图书馆 CIP 数据核字(2017)第 129957 号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：北京天宇星印刷厂
装　　订：北京天宇星印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：15　字数：314 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

作者介绍

Fabrizio Soppelsa 是一家 OpenStack 公司——Mirantis 的高级工程师。从 Docker 0.3 版本开始，他就是 Docker 的积极使用者和倡导者，他用三个国家的语言发表了 Docker 工具相关的多篇文章。他也是一些项目，特别是 Machine 项目的实际贡献者。他目前生活在俄罗斯的莫斯科，他和他的蜘蛛 Mosha 是那里 Docker 见面会的组织者。

我要感谢 ClusterHQ 的工作人员对 Flocker 的帮助，特别感谢 Ryan Wallner。也要感谢 Yandex 团队和 Denis Kutin 提供了免费的 OpenStack 实验室，让我能够很容易地使用。感谢 Mirantis 创建了（我认为是）最好的 OpenStack 发行版本。感谢 Docker 团队和 Docker 社区带给我的所有快乐。

Chanwit Kaewkasi 是泰国苏兰拉里理工大学计算机学院的助理教授。Chanwit 从 0.1 版本就开始参与 Docker Swarm 项目的贡献，他协同设计并且实现了策略过滤器、ZooKeeper 发现，以及其他特性。他目前是 Docker Swarm 的维护者以及 Docker Captain（社区给 Docker 专家的称号）。

我还要感谢我的妻子——Pitchaya，感谢她的鼓励以及对我的工作，包括这本书的大力支持。

*要送给 Docker Engineering 团队特别的感谢，感谢他们开发出的伟大的软件。*感谢苏兰拉里理工大学为我提供了绝佳的工作场所。感谢我的父母对我的支持。最后还要感谢 Fabrizio 邀请我合作撰写本书。

审阅者介绍

Baohua Yang 是 IBM 的资深研究员。他的兴趣包括云计算、Fintech、分布式系统和分析的核心领域。他对那些新兴技术特别感兴趣，比如，SDN/NFV、容器、大数据、区块链和认知计算。

作为首席架构师，他领导企业产品的架构设计和系统实现，并且帮助解决了行业解决方案的关键技术难题。

作为开源社区的贡献者，他向数个项目提交代码、方案和演讲，包括 OpenStack、Hyperledger、OpenvSwitch、Docker、OpenDaylight 和 Kubernetes，并且领导了一些项目，包括 easyOVS、Hyperledger Fabric-SDK-py 和 Cello。他现在是中国 Hyperledger 技术工作组的主席。

他在顶尖的互联网会议和期刊（包括 IEEE INFOCOM，IEEE Trans on Computers）上发表了十多篇文章，并且参与撰写了一些技术书籍和专利。他现在是数个学术会议和期刊的 TPC 成员。

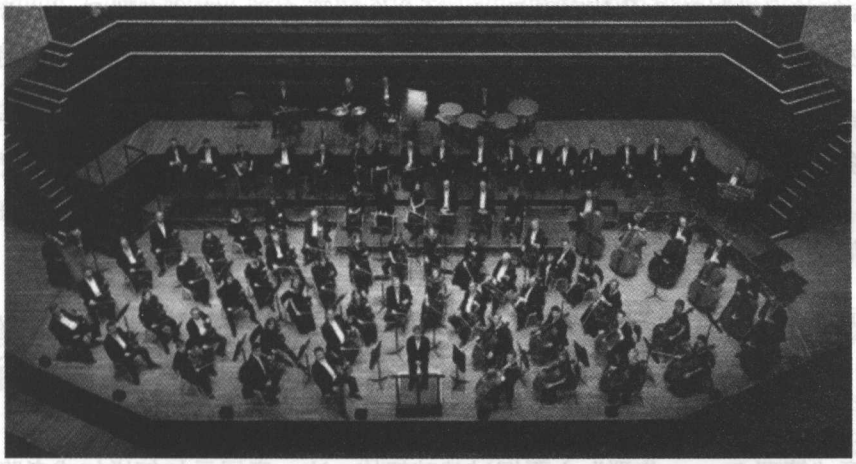
他的主页：https://yeasy.github.com。

前言

欢迎来到《Swarm 容器编排与 Docker 原生集群》一书！这是一本关于容器和分布式系统的书。本书将介绍如何使用原生的 Docker 工具建模微服务、生成任务、扩大应用程序的规模，以及将容器推送到 Docker 集群里！一句话来说，本书将讨论 Docker 的编排。

随着最近 Swarm Mode 的崛起，以及 Docker Engine 启用了 Swarm 功能，编排 Docker 的最佳方式其实还是 Docker！

听上去不错，但是“编排 Docker”是什么意思呢？什么是编排？更确切的说法是，什么是管弦乐队？



管弦乐队指的是音乐家的全体，它由指挥家指挥，指挥家负责控制节奏、旋律，塑造出音乐的整体。弦乐队、管乐队、打击乐队、键盘乐队以及其他乐队都会遵循指挥家的指挥，共同演奏出惊人的交响乐曲，比如贝多芬的《第九交响乐》。

类似地，在容器编排系统里，音乐家是任务，指挥家则是领导者服务（Swarm primitives）。任务并不演奏旋律，或者并不仅仅做这些：更为抽象地说，它们执行一些计算型工作，比如，运行 Web 服务器。而指挥家——Swarm，则负责它们的预配，它们的可用性，它们的

链接，它们的扩展。这也就是大家所说的“Docker 编排”。

本书讲述如何预配这样的 Docker“管弦乐队”，如何保证服务的可用性，如何连接任务，以及如何扩展平台，从而演奏出属于应用程序的动人交响乐。

本书范围

第 1 章“欢迎来到 Docker Swarm”会介绍 Swarm，并且解释用户为什么需要集群解决方案来管理容器。这一章介绍 Swarm 的特性，介绍其架构的高层级描述。这一章还设计了一些示例，讲述 Swarm 和 Fleet、Kubernetes、Mesos 的不同之处。之后也会介绍 Docker 工具的安装以及两种 Swarm 的预配方式：本地的 Swarm Standalone 和远程在 DigitalOcean 上的 Swarm Mode 集群。

第 2 章“探索发现服务”是描述性语言最多、最抽象的一章。这一章介绍发现机制和共识算法是什么，以及它们为什么对于分布式系统来说至关重要。本章会详细介绍 Swarm Mode 包含的共识机制 Raft 及其实现 Etcd。还会介绍第 1 章“欢迎来到 Docker Swarm”里所使用的发现机制的局限性，并且使用 Consul 扩展上一章的本地示例，之后重新将其部署。

第 3 章“遇见 Docker Swarm Mode”介绍全新的 Docker kit，它能够帮助用户创建任何规模的任务集群。本章会介绍 Docker Swarm Mode 的基础——SwarmKit，介绍它在 Docker 1.12+版本里是如何工作的，讨论其架构、理念，它和“旧”Swarm 的不同之处在哪里，以及它是如何通过抽象出服务和任务来组织工作负载的。

第 4 章“创建生产级别 Swarm”介绍并且讨论了社区驱动的项目——Swarm2k 和 Swarm3k，我们实验了 2300 和 4800 个节点的 Swarm 集群，可以运行成千上万个容器。最后总结了可用计划，预配多大规模的集群，以及实验中的经验教训。

第 5 章“管理 Swarm 集群”主要探讨基础架构。这一章展示如何增加或者降低 Swarm 的规模，如何 promote 以及 demote 节点，以及如何更新集群和节点的属性。这一章还会介绍 Shipyard 和 Portainer.io，其可以作为 Swarm 的图形 UI。

第 6 章“Swarm 上真实应用的部署”介绍了将真实应用程序放到 Swarm 上，并且讨论了 Compose、Docker Stacks 和 Docker Application Bundles。这一章展示了典型的部署工作流如何在集群里过滤并且调度容器，将其作为服务启动，将容器作为任务处理。这一章从

定义一个使用 Nginx 的 Web 服务开始，然后部署一个使用 MySQL 的 WordPress，最终介绍一个更为实际的应用：Apache Spark。

第 7 章“平台的向上伸缩”将在前几章的基础上开始新的话题讨论。这一章将介绍 Flocker，给 Swarm 上运行的 Spark 增加存储能力，并且会展示如何安装，以及如何和 Swarm 一起大规模、自动地使用它。这一章将完善之前的 Spark 示例，运行一些真实的大数据 job，并且为该基础架构搭建基础的监控系统。

第 8 章“Swarm 附加特性的探索”讨论了一些对于 Swarm 来说很重要的高级话题，包括 Libnetwork 和 libkv。

第 9 章“Swarm 集群和 Docker 软件供应链的安全加固”关注 Swarm 集群的安全方面，会介绍平台的参数、证书、防火墙等概念，并且会介绍 Notary。

第 10 章“Swarm 和云”介绍了在云供应商那里运行 Swarm 的最为流行的几种方案。将在 AWS 和 Azure 上安装 Swarm，然后介绍 Docker Datacenter，最后会转向 OpenStack，介绍在 Magnum 上如何安装以及管理 Swarm，Magnum 是 OpenStack 提供的容器即服务方案。

第 11 章“Swarm 的未来展望”展望了 Docker 编排的趋势，比如软件定义的基础架构、Infrakit、unikernel 以及 Caas。伟大的征途尚未结束！

阅读本书的要求

本书假定读者有在命令行里使用 Docker 的经验：本书通篇会持续地拉取镜像、运行容器、定义服务、暴露端口以及创建网络。

另外，读者最好对网络协议有一些基本了解，并且熟悉公有云和私有云的概念，比如虚拟机和 tenant 网络。

要实践本书的示例，读者需要 Docker 及其工具。第 1 章“欢迎来到 Docker Swarm”介绍了它们的安装方式。

另外，要想完全学习到示例里的知识，读者还需要能够访问一种公有云（比如 AWS，Azure 或者 DigitalOcean）或者私有云（比如 OpenStack）来初始化出虚拟机。

目标读者

本书写给 Docker 的用户——开发人员和系统管理员，那些想要利用现有的 Swarm 和 Swarmkit 的功能，借助容器大幅扩展应用程序的人们。

约定的格式

本书使用了一系列文本格式来区分不同类型的信息。这里列出了这些格式的示例，并且解释了其含义。

文本里的代码、数据库表名、文件夹名、文件名、文件扩展、路径名、URL、用户输入，以及 Twitter 处理器格式如下：“当执行 docker swarm init 时，只需复制和粘贴所输出的行”。

代码如下所示：

```
digitalocean:
      image: “docker-1.12-rc4”
      region: nyc3
      ssh_key_fingerprint: “your SSH ID”
      ssh_user: root
```

命令行输入或输入如下所示：

```
      # Set $GOPATH here
      go get https://github.com/chanwit/belt
```

新的术语以及**重要单词**会加重表示。屏幕上可以看到的文本，比如菜单或者对话框，显示为这样的样式：“UI 有预期选项，包括启动容器的一系列模板，比如 **MySQL** 或者**私有 Registry**，但是撰写本书时还不支持 Swarm 服务”。

警告或者重要提示使用这个图标。

建议或技巧使用这个图标。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31792

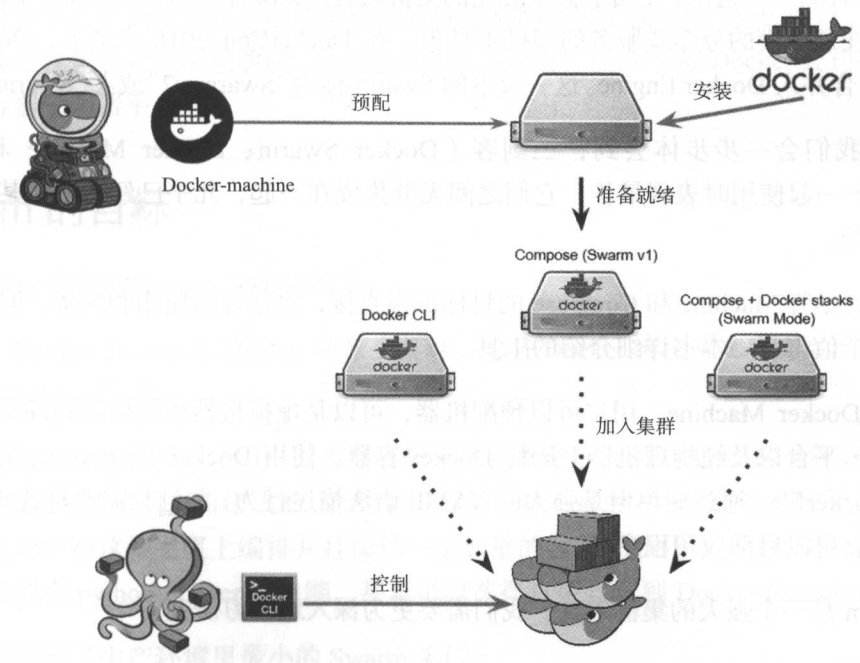
目录

第 1 章

欢迎来到 Docker Swarm

Docker 是如今公认的最受关注的开源技术之一。原因也很简单，Docker 让所有人都可以使用容器技术，它自带可移动的天性，并且有活跃社区的强大支撑。

在早期，用户开始使用 Docker 是因为它是易用的工具，帮助用户解决了很多难题，如拉取、打包、隔离，并且让应用程序可以跨系统地无缝迁移。



简化的Docker生态系统

在上图中我们可以看到一大群鲸鱼之间合作良好。但是，随着容器的到来，大家开始寻找可以高效编排大量容器的工具。Docker 团队尝试解决这个问题，他们在 2015 年发布了 Docker Swarm，下文简称 Swarm，以及 Docker Machine 和 Docker Compose，它们是 Docker 生态系统的一部分。上图展示了简化的 Docker 生态系统，包括由 Docker Machine 预配新的可以使用 Docker 的机器，然后一系列机器将被编成 Docker Swarm 集群。之后就可以使用 Docker Compose 来向集群里部署容器，只要它使用的是常规的 Docker Engine。

为 Docker 构建原生的集群管理工具的计划早在 2014 年初就开始了，当时作为一个通信协议项目，称为 *Beam*。之后，它被实现为一种后台程序，使用 Docker API 来控制异构化的分布式系统。项目重命名为 `libswarm`，`Swarmd` 是其后台程序。项目保持了之前的理念，允许任何 Docker 客户端连接到 Docker Engine 池里。该项目的第三代被重新进行设计，使用相同的 Docker Remote API 集，并且在 2014 年 11 月份重命名为“Swarm”。基本上，Swarm 最重要的部分就是其远程 API；维护人员通过努力工作，让这些 API 和 Docker Engine 的所有版本 100%兼容。我们称 Swarm 第一代为“Swarm v1”。

2016 年 2 月份，核心团队发现中央服务的扩展被限制之后，Swarm 在内部被再次重新设计为 `swarm.v2`。这次，使用了去中心化的集群设计。2016 年 6 月份，发布了 SwarmKit，可以作为任意规模的分布式服务的编排工具包。在 DockerCon 2016 大会上，Docker 宣布 SwarmKit 合并入 Docker Engine。这一版本的 Swarm 称为“Swarm v2”或者“Swarm mode”。

之后我们会一步步体会到，三剑客（Docker Swarm、Docker Machine 和 Docker Compose）一起使用时表现最佳，它们之间无缝集成在一起，几乎已经无法将某一个当成单独的部分。

但是，尽管 Machine 和 Compose 的目标都很直接，也很容易使用和理解，但是 Swarm 的确是一个值得写一本书详细介绍的工具。

使用 Docker Machine，用户可以预配机器，可以是虚拟机器也可以是物理机器，并可以在若干云平台以及纯物理机器上运行 Docker 容器。使用 Docker Compose，用户可以快速定义 Dockerfile，通过简单但是强大的 YAML 语法描述行为，并且只需要将这些文件“组合”起来就可以启动应用程序。

Swarm 是一个强大的集群工具，我们需要更为深入地学习研究。

本章将介绍如下主题：

- 什么是容器的编排
- Docker Swarm 的基础和架构
- 和其他开源编排工具的不同之处
- “老”Swarm，Swarm v1
- “新”Swarm，Swarm Mode

集群工具和容器管理器

集群工具是一种软件，允许运维人员和单个终端沟通，其可以向一系列资源（本书里就是容器）发送命令并且编排这些资源。取代在集群上手动分发工作负载（容器）的方式，集群工具用来自动化这些任务以及其他很多任务。集群工具决定在哪里启动 job（容器），如何存储 job，什么时候最终重启 job 等。运维人员仅仅需要配置一些行为，决定集群的拓扑和规模、调优设置，并且启用或者停用高级特性。Docker Swarm 就是这样一种容器的集群工具。

除了集群工具，我们还可以选择容器管理器平台。它们不提供容器托管，但是它们和一个或者多个已有系统交互；这样类型的软件通常都提供了很好的 Web 接口、监控工具，以及其他可视化或者高层级功能。Rancher 或者 Tutum（2015 年被 Docker 公司收购）就是这样的容器管理器平台。

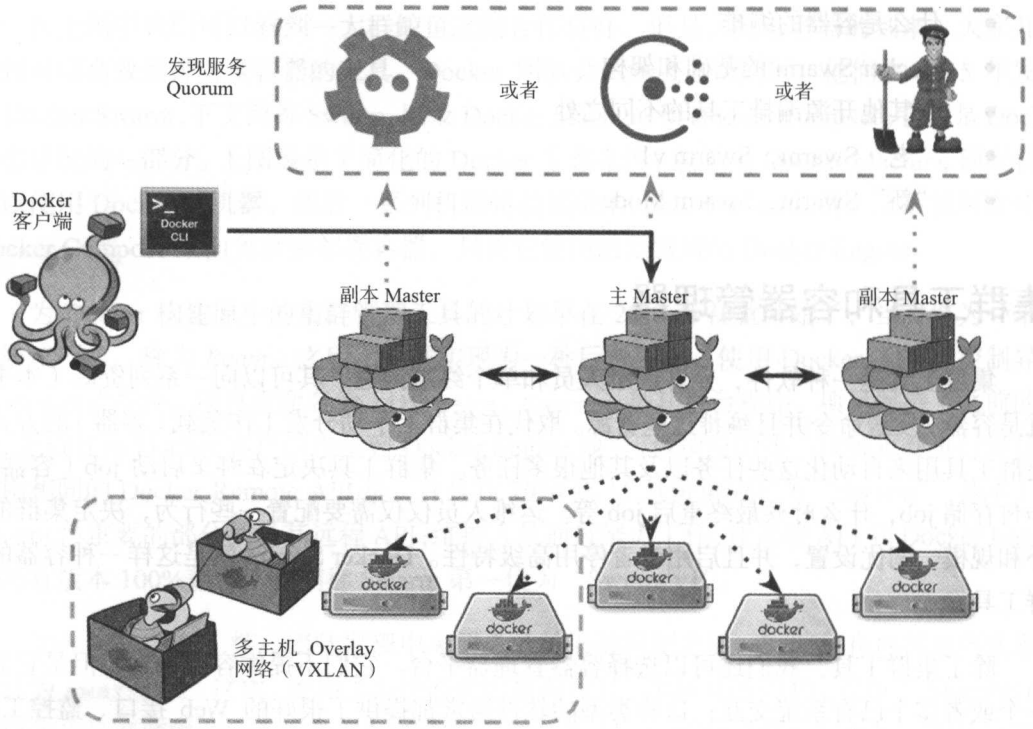
Swarm 的目标

Docker 自身对 Swarm 的描述如下：

> Docker Swarm 是 Docker 的原生集群。它将 Docker 宿主机池转变成单个虚拟的 Docker 宿主机。

Swarm 是一种工具，让用户以为自己管理的是单个巨大的 Docker 宿主机，而这个宿主机是由很多 Docker 宿主机组成的。这些主机看上去是一体的，并且使用一个命令行入口点。Swarm 让用户在这些主机上编排并且操作一定数量的容器，使用常规的 Docker 工具、Docker 原生工具或者 python-docker 客户端，甚至可以选择直接 curl 到 Docker Remote API 上。

下图展示了生产环境里最小的 Swarm 集群：



为什么使用 Swarm

使用容器集群解决方案有很多原因。随着应用程序的壮大成熟，我们会遇到很多全新的需求，比如可扩展性、可管理性以及高可用性。

市场上有很多可用的工具，使用 Docker Swarm 可以带来如下优势。

- **原生集群**：Swarm 是 Docker 原生的，由 Docker 团队和社区开发。它最初的创造者是 Andrea Luzzardi 和 Victor Vieux，他们是 Docker Engine Remote API 的早期实现者。不需要任何额外的需求，Swarm 就可以和 Machine、Compose 以及生态系统里的其他工具集成。
- **生产环境可用**：Swarm v1 在 2015 年 11 月份成熟，并且可以在生产环境里使用。研发团队已经证明 Swarm 能够扩展到控制 1000 个节点的 Engine。Swarm v2 因为使用了去中心化的发现机制，允许构造数千个节点的集群。

- **开箱即用**：Swarm 不要求用户重新架构自己的应用程序来适配其他的编排工具。用户可以使用自己的 Docker 镜像和配置，不需要任何改动，就可以实现大规模部署。
- **易于搭建和使用**：Swarm 很容易操作。仅仅通过在 Machine 命令里添加一些参数，或者使用 Docker 1.12 版本后的 Docker 命令，就可以实现高效部署。将发现服务集成到 Swarm Mode 里，使其可以快速安装——不需要搭建外部的 Consul、Etcd 或者 Zookeeper 集群。
- **活跃的社区**：Swarm 是一个有活力的项目，有非常积极的社区支撑，开发很快速。
- **Hub 上可用**：用户不需要安装 Swarm，它是一个 Docker 镜像（Swarm v1），因此用户仅仅需要从 Hub 里拉取这个镜像就可以运行，或者它已经集成进 Docker Engine。Swarm Mode 已经集成进 Docker 1.12+版本了。

真实的示例

下面这些项目选择使用 Docker Swarm。

- Rackspace Carina 是基于 Docker Swarm 构建的：Rackspace 提供托管的容器环境，内部是基于 Docker Swarm 的。
- Zenly 使用 Swarm，并且集群跨 Google Cloud Platform 和纯物理服务器。
- ADP 使用 Docker 和 Swarm，提高了遗留系统部署的速度。
- Swarm 可以使用 Amazon AWS 和 Microsoft Azure 公有云上直接可用的模板来完成部署。

宠物模型 vs 牛群模型

在创建并且利用基础架构时有两种对立的方案：宠物模型 vs 牛群模型。

在宠物模型里，管理员部署服务器或者虚拟机，或者本书中的容器，并且对其持续进行维护。她/他登录机器或容器，安装软件，完成配置，并且确保一切运转正常。因此，这些机器或者容器是她/他的宠物。

相对应地，当管理员把基础架构当成牛群模型时，他们并不关心基础架构某个组件的命运。她/他不会登录到任何单元里，也不会做手动处理。相反，他们使用批量方案，借助

自动化工具完成部署、配置以及管理。如果某个服务器或者容器死机了，它会被自动复活，或者生成另一个服务器或容器来替代有问题的组件。因此，在这种场景下，运维人员管理的是牛群模型。

本书的第 1 章会使用宠物模型来向读者介绍一些基础概念。但是之后开始讨论真实应用时，就会使用牛群模型。

Swarm 特性

前文已经定义了 Swarm 的主要目的，但它是如何实现这些目标的呢？ Swarm 的核心特性如下：

- Swarm v1 支持 1.6.0 版本或更新版本的 Docker Engine。从 1.12 版本开始，Swarm v2 内嵌到了 Docker Engine 里。
- Swarm 每个版本的 API 都和同版本的 Docker API 兼容。API 可以向后兼容一个版本。
- 在 Swarm v1 里，为了多个 Swarm 主节点的实现，选主机制使用的是主库（仅仅在使用发现服务，比如 Etcd、Consul 或者 ZooKeeper 部署 Swarm 时才支持）。
- 在 Swarm v2 里，使用去中心化的机制来构建选主机制。Swarm v2 不再需要特定的发现服务，因为它集成了 Etcd，这是 Raft 公式算法的一种实现（详见第 2 章“探索发现服务”）。
- 在 Swarm v1 的术语里，主 Swarm master 节点被称为 primary（主节点），其他 master 节点被称为 replica（副本）。在 Swarm v2 里，有 Master 和 Worker 节点的概念。集群使用 Raft 自动管理主节点。
- 基础和高级调度选项。schedule（调度器）是一种决定容器物理上放置于哪台主机的算法。Swarm 使用一系列内置的调度器。
- Constraint（约束条件）和 affinity（共同关系）辅助运维人员做出调度决策。比如，某个用户想要让数据库容器物理上接近，就可以通知调度器去这么做。约束条件和共同关系使用 Docker Swarm label（标签）。
- 在 Swarm v2 里，使用内建 DNS Round-Robin 实现集群内的负载均衡，也支持外部负载均衡，通过路由网格机制，由 IPVS 实现。

- 高可用和故障恢复机制意味着用户可以创建超过一个 master 的 Swarm。因此，如果某个 master 服务宕机了，还有其他 master 可以继续控制。当构建至少 3 个节点的集群时，默认可用 Swarm v2。所有节点都可以作为 master 节点。另外，Swarm v2 包括健康指标信息。

类似项目

除了 Docker Swarm，还有一些其他方式可以集群化容器。为了讨论的完整性，在深入介绍 Swarm 之前，这里先简要介绍最为流行的其他几种开源解决方案。

Kubernetes

Kubernetes（http://kubernetes.io），也称为 k8s，其和 Docker Swarm 的目标是一致的；也是一种容器集群的管理器。Kubernetes 起源于 Google 实验室的 Borg 项目，之后被开源并且在 2015 年发布了稳定版本，支持 Google Cloud Platform、CoreOS、Azure 和 vSphere。

到目前为止，Kubernetes 是在 Docker 里运行容器，通过称为 Kubelet 的 API 发指令，Kubelet 是注册并且管理 Pod 的服务。从架构上来看，Kubernetes 在逻辑上将其集群分为 Pod，而不是纯容器。Pod 是最小的部署单元，物理上代表由一个或者多个容器组成的应用程序，这些容器通常部署在一起，共享一些资源，比如存储和网络，用户可以使用 Compose 在 Docker 里模拟 Pod，从 Docker 1.12 版本开始可以创建 Docker DAB（Distributed Application Bundle，分布式应用程序组）。

Kubernetes 包括一些基本的集群特性，比如标签、监控检查器、Pod 注册表，有可配置的调度器，以及类似 ambassador 或者负载均衡器这样的服务。

实际上，Kubernetes 的用户利用 kubectl 客户端和 Kubernetes master 交互，master 是集群的控制单元，命令 Kubernetes 节点完成工作，这些节点称为 Minion。Minion 运行 Pod，所有组件通过 Etcd 粘合在一起。

在 Kubernetes 节点上，用户可以看到一个运行着的 Docker Engine，它运行着一个 kube-api 容器，以及一个称为 `kubelet.service` 的系统服务。

kubectl 的命令非常直观，比如：

- `kubectl cluster-info`，`kubectl get pods` 以及 `kubectl get nodes`，可以获得集群及其健康状况的信息。
- `kubectl create -f cassandra.yaml` 及其衍生的 Pod 命令，用来创建、管理以及销毁 Pod。
- `kubectl scale rc cassandra --replicas=2` 用来扩展 Pod 和应用程序。
- `kubectl label pods cassandra env=prod` 用来配置 Pod 标签。

这仅仅是 Kubernetes 高层级的简要介绍。Kubernetes 和 Docker Swarm 的主要区别在于：

- Swarm 的架构更为直接，更容易理解。Kubernetes 则需要更多的精力去学习其基本知识。但是学习总是好事嘛！
- 架构上的另一点不同——Kubernetes 基于 Pod，Swarm 基于容器以及 DAB。
- 用户需要安装 Kubernetes。要么是在 GCE 上使用 CoreOS 部署，要么在 OpenStack 上部署，用户必须自己完成安装。当必须部署和配置 Kubernetes 集群时，需要一些额外的工作。Swarm 则集成到了 Docker 里，不需要任何额外的安装。
- Kubernetes 有一个额外的概念——Replication Controller（复制控制器），这是一种技术，确保由一些模板所描述的所有 Pod 在指定时间里是一直运行着的。
- Kubernetes 和 Swarm 都使用了 Etcd。但是 Kubernetes 里 Etcd 是外部辅助服务的，而 Swarm 里是集成并且运行在管理器节点上的。

Kubernetes 和 Swarm 的性能比较可能会挑起“圣战”，本书只列举事实数据。一些基准测试显示 Swarm 启动容器很快，而另外一些测试显示 Kubernetes 运行工作负载非常快。本书认为对于基准测试结果应该持保留态度。也就是说，Kubernetes 和 Swarm 都适合运行大型、快速并且可扩展的容器集群。

CoreOS Fleet

Fleet（https://github.com/coreos/fleet）是另一种容器编排器的可选方案。它来自于 CoreOS 容器产品家族（包括 CoreOS、Rocket 和 Flannel），和 Swarm、Kubernetes 以及 Mesos 有着本质的不同，它从架构上看就是作为系统的扩展而存在的。Fleet 通过调度器操作，跨集群节点分发资源和任务。因此，其目标不仅仅是提供一个纯容器的集群，而是要成为一个分布式的、更为精妙的系统。比如，可以在 Fleet 之上运行 Kubernetes。

Fleet 集群包括负责调度 job 的引擎、其他管理操作，以及运行在每台主机上的代理，

它们负责物理上执行所分配的 job，并且持续向引擎汇报状态。Etcd 是发现服务，将所有东西粘合在一起。

用户通过主命令 `fleetctl` 和 Fleet 集群交互，可以列举、开始以及停止容器和服务。

因此，总的来说，Fleet 和 Docker Swarm 的不同点在于：

- 它是分发任务的更高层级的抽象，并不仅仅是容器编排器。
- Fleet 不仅仅是集群的分布式初始化系统。Systemd 是针对一台主机的，而 Fleet 是针对主机集群的。
- Fleet 集群化一堆 CoreOS 的节点。
- 可以在 Fleet 之上运行 Kubernetes，从而利用 Fleet 的弹性和高可用特性。
- 没有稳定和健壮的方式自动集成 Fleet 和 Swarm v1。
- 目前，Fleet 并没有试验过运行超过 100 个节点、1000 个容器的集群（https://github.com/coreos/fleet/blob/ master/Documentation/fleet-scaling.md），而 Swarm 可以运行 2300 个节点，之后更是达到 4500 个节点。

Apache Mesos

如果将 Fleet 看作集群的分布式初始化系统，那么就可以将 Mesos（https://mesos.apache.org/）看作分布式内核。使用 Mesos，用户就可以将所有节点资源看成一个整体，然后在其上运行容器集群。

Mesos，起源于 2009 年的伯克利大学，是一个成熟的项目，并且已经有生产环境上的成功经验，比如 Twitter。

它的目标比 Fleet 更宽泛，支持多平台（用户可以在 Linux、OS X 和 Windows 节点上运行），并且能够运行异构的 job。用户可以同时在 Mesos 上运行容器集群，纯粹的大数据 job（Hadoop 或者 Spark）以及其他 job，包括持续集成、实时计算、Web 应用程序、数据存储等。

Mesos 集群由一个 Master、多个 slave 和框架组成。Master 向 slave 分配资源和任务，它负责系统通信，并且运行发现服务（ZooKeeper）。框架是什么呢？框架是应用程序。一个框架包括调度器和执行器，前者分发任务，后者执行任务。

通常容器通过一个名为 Marathon（https://mesosphere.github.io/marathon/docs/native-docker.html）的框架在 Mesos 上运行。

比较 Mesos 和 Docker Swarm 的区别意义不大，因为它们能够互补地共同运行，也就是说，Docker Swarm v1 可以在 Mesos 上运行，Swarm 源代码里的一部分就是专门实现这个场景的。而 Swarm Mode 和 SwarmKit 则和 Mesos 非常类似，因为它们将 job 抽象成任务，并且将任务分组成服务，从而实现集群里的负载分发。本书会在第 3 章“遇见 Docker Swarm Mode”里更为详细地讨论 SwarmKit 的特性。

Kubernetes vs Fleet vs Mesos

Kubernetes、Fleet 和 Mesos 试图解决类似的问题；它们提供资源的抽象层，并且允许用户和集群管理器交互。随后用户可以启动 job、任务和项目。不同之处在于所提供的“开箱即用”的特性，以及用户在分配、扩展资源和 job 上可以自定义灵活度的大小。这三者之中，Kubernetes 更为自动化；Mesos 可自定义程度更强，从某个角度看，更为强大（如果你需要这样的强大的话）。

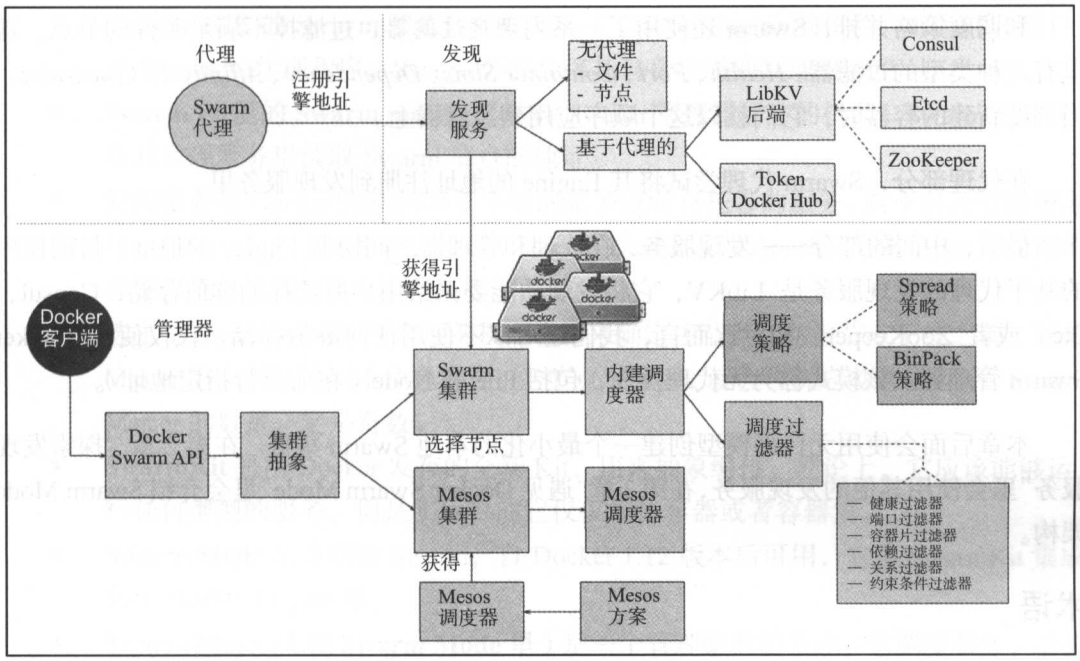
Swarm vs 所有

那么，应该使用哪种方案呢？当用户遇到某个问题时，开源的力量将足够包容，让用户可以使用多种可能互相交叉的技术，达成最终的目标。问题是如何选择以及选择哪种方案来解决问题。Kubernetes、Fleet 和 Mesos 都很强大，都是很有意思的项目，Docker Swarm 也是一样的。

如果考量的是这四者当中哪个自动化更强、更易于理解的话，Swarm 胜出。当然这并不总是其优势所在，不过本书会展示 Docker Swarm 是如何帮助用户完成实际工作的，记住 DockerCon 上 Solomon Hykes、Docker 的 CTO 和创始人所给出的建议：Swarm 是能够提供通用接口的层，该层可连接到很多编排和调度框架上。

Swarm v1 架构

本节讨论 Docker Swarm 的整体架构。Swarm 的内部结构如下图所示。



Docker Swarm v1的内部结构

先从**管理器**部分开始介绍，图片的左侧有一个标为 *Docker Swarm API* 的长方形。如前文所述，Swarm 暴露出一系列和 Docker 类似的远程 API，用户可以使用任意 Docker 客户端连接到 Swarm 上。但是，Swarm API 和标准的 Docker 远程 API 略有不同，因为 Swarm API 还包含集群相关的信息。比如，在 Docker Engine 上运行 `docker info` 可以得到单个 Engine 的信息，但是如果在 Swarm 集群上调用 `docker info`，则会得到集群里的节点数量以及每个节点的信息和健康状况。

Docker Swarm API 旁边的长方形是*集群抽象*。这是一个抽象层，允许不同类型的集群实现为 Swarm 的后端，并且共享同一组 Docker 远程 API。目前有两种集群后端实现：内建 Swarm 集群实现和 Mesos 集群实现。Swarm *集群*和*内建调度器*的长方形表示内建的 Swarm 集群实现，而标识为 Mesos *集群*的长方形则表示 Mesos 的集群实现。

Swarm 后端的*内建调度器*包含一些*调度策略*。两种策略分别是 *Spread* 和 *BinPack*，我们会在后面的章节详细介绍。如果你对 Swarm 比较熟悉，就会注意到这里缺失了 Random 策略。Random 策略没有包含在这里是因为其仅作为测试用途。

和调度策略并排，Swarm 还使用了一系列调度过滤器，过滤掉不满足条件的节点。这里有六种类型的过滤器：*Health*、*Port*、*Container Slots*、*Dependency*、*Affinity* 和 *Constraint*。当调度新建的容器时，它们就以这个顺序应用到过滤器上。

在**代理**部分，Swarm 代理尝试将其 Engine 的地址注册到发现服务里。

最后，中间的部分——**发现服务**，在代理和管理器之间协调 Engine 的地址。目前使用的基于代理的发现服务是 LibKV，它将发现功能委派给用户所选择的键值存储，Consul、Etcd 或者 ZooKeeper。相对比而言，我们还可以不使用任何键值存储，仅仅使用 Docker Swarm 管理器。该模式称为无代理发现，包括 File 和 Node（在命令行指定地址）。

本章后面会使用无代理模型创建一个最小化的本地 Swarm 集群。在第 2 章“探索发现服务”里会使用其他的发现服务，在第 3 章“遇见 Docker Swarm Mode”里会介绍 Swarm Mode 架构。

术语

在开始其他小节之前，这里先回顾一些 Docker 相关的术语，回忆一下 Docker 的理念并且介绍 Swarm 的关键词。

- **Docker Engine** 是运行在宿主机器上的 Docker daemon。本书有时候简称为 Engine。用户通常通过 systemd 或者其他启动服务调用 `docker daemon` 来启动 Engine。
- **Docker Compose** 是一个工具，以 YAML 的格式描述多容器服务的架构方式。
- **Docker stack** 是创建多容器的应用程序（由 Compose 描述），而不是单个容器的二进制结果。
- **Docker daemon** 和 Docker Engine 这两个术语意思一样，可以互相替换。
- **Docker 客户端**是打包在同一个 docker 可执行文件里的客户端程序。比如，当运行 `docker run` 时，使用的就是 Docker 客户端。
- **Docker 网络**是软件定义的网络，将一系列容器链接到同一个网络里。默认使用 Docker Engine 自带的 libnetwork（https://github.com/docker/libnetwork）实现。但是用户可以使用插件选择部署第三方的网络驱动。
- **Docker Machine** 是一个工具，用来创建能够运行 Docker Engine 的称为**机器**的主机。

- **Swarm v1 里的 Swarm 节点**是预先安装了 Docker Engine 的机器，并且同时运行着 Swarm 的代理程序。Swarm 节点会将其自身注册到发现服务里。
- **Swarm v1 里的 Swarm master** 是运行 Swarm 管理器程序的机器。Swarm master 从其发现服务里读取 Swarm 节点的地址。
- **发现服务**是 Docker 提供的基于令牌的服务或者自托管的服务。对于自托管的服务，用户可以运行 HashiCorp Consul、CoreOS Etcd，或者 Apache ZooKeeper 作为键值存储来提供发现服务。
- **选主**是 Swarm Master 所实现的机制，用来确定主节点。在主节点下线之前，其他 Master 节点都作为副本角色存在，主节点下线后，就会再次启动选主程序。Swarm Master 的数量一定是奇数。
- **SwarmKit** 是由 Docker 发布的全新 Kit，用来抽象编排。理论上，它应该能够运行任何类型的服务，但是实际目前它仅仅编排容器或者容器集。
- **Swarm Mode** 是全新的 Swarm，自 Docker 1.12 版本后可用，它将 SwarmKit 集成到了 Docker Engine 里。
- **Swarm Master（在 Swarm Mode 里）**是一个管理集群的节点：它调度服务，维护集群的配置（节点、角色和标签），并且确保仅仅存在一个集群领导者。
- **Swarm Worker（在 Swarm Mode 里）**是一个运行任务的节点，运行如托管容器任务。
- **服务**是工作负载的抽象。比如，用户可以让服务“nginx”复制 10 次，意味着用户将有 10 个任务（10 个 nginx 容器）分布在集群上，并且由 Swarm 自身做负载均衡。
- **任务**是 Swarm 的工作单元。一个任务就是一个容器。

开始使用 Swarm

本节开始安装两个小型的 Swarm v1 和 v2 的集群，用来做概念论证，第一个集群搭建在本地，第二个集群搭建在 Digital Ocean 上。为了能够顺利执行所有命令，请检查如下前提条件列表，确保满足了所有条件，然后就可以开始了。

要完成本节示例，需要：

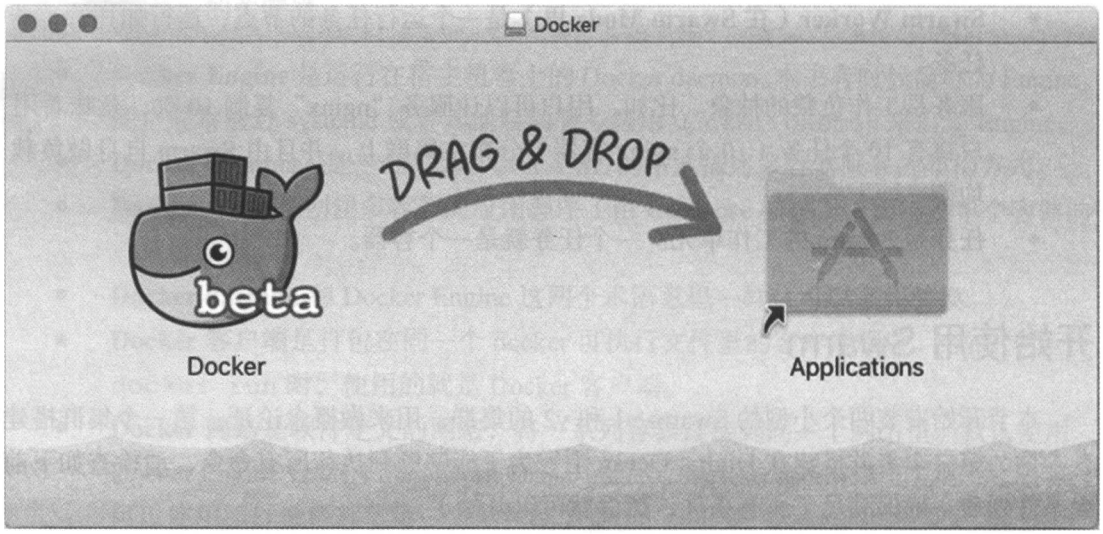
- 一台 Windows、Mac OS X 或者 Linux 系统的电脑。

- Bash 或者 Bash 兼容的 shell。在 Windows 系统上，可以使用 Cygwin 或者 Git Bash。
- 本地集群示例需要安装 VirtualBox 的最新版本。
- 本地集群示例至少需要 4GB 内存提供给 4 个 VirtualBox 的实例，每个实例需要 1GB 内存。
- Docker 客户端，Swarm v1 要不低于 1.6.0 版本，Swarm v2 要不低于 1.12 版本。
- Docker Machine 的最新版本，目前是 0.8.1。

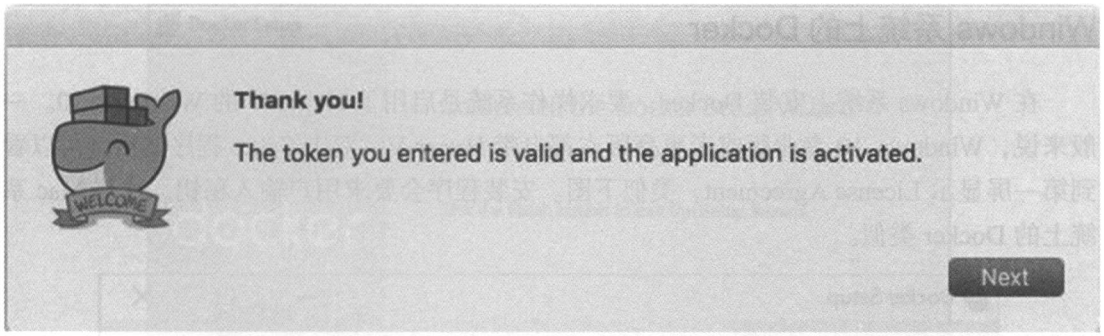
Mac 系统上的 Docker

Docker 在 2016 年年初发布了 Mac 系统上的 Docker 和 Windows 系统上的 Docker 的版本。它比 Docker Tookbox 更好，因为它包含了用户所需的 Docker CLI 工具，但是并不再使用 boot2docker 和 VirtualBox（而是使用 unikernel，本书会在第 11 章“Swarm 未来展望”里详细介绍），并且它被彻底集成进了操作系统（Mac OS X Sierra 或者启用 Hyper-V 的 Windows 10）。

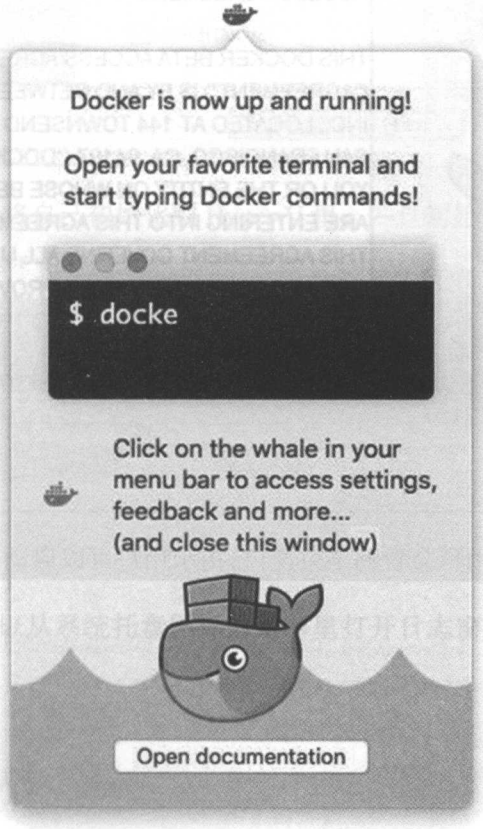
用户可以从 https://www.docker.com/products/overview#/install_the_platform 处下载 Docker 桌面，安装很简单。



如果使用的是 Mac OS X 系统的话，只需要将 Docker 的 beta 图标拖曳到应用程序文件夹里，输入 beta 注册码就可以了。

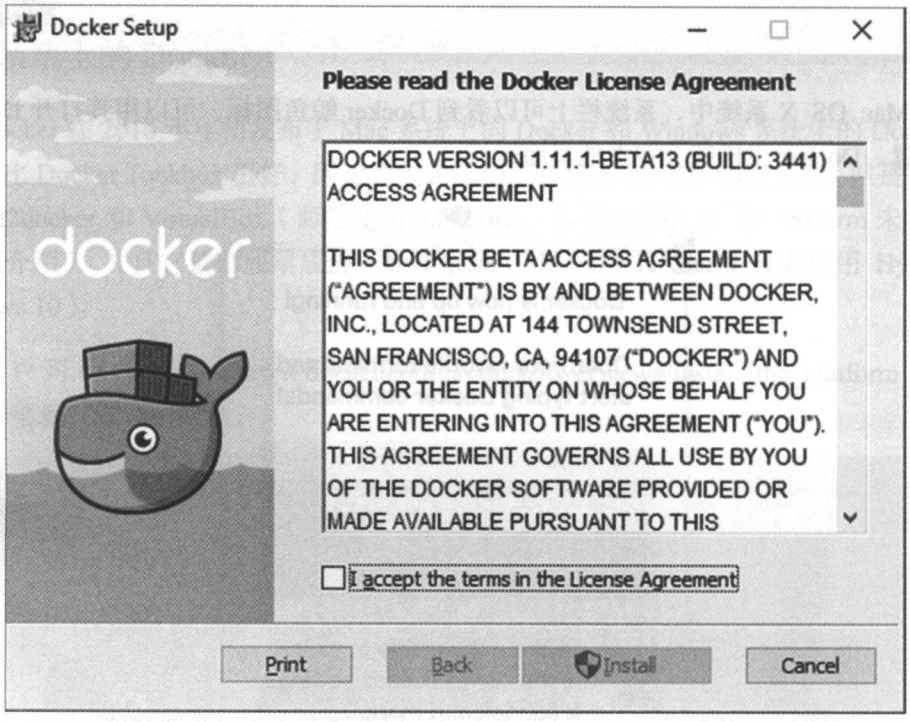


在 Mac OS X 系统中，系统栏上可以看到 Docker 鲸鱼图标，可以用其打开 Docker 并完成设置。Docker 主机就会在本地桌面运行。

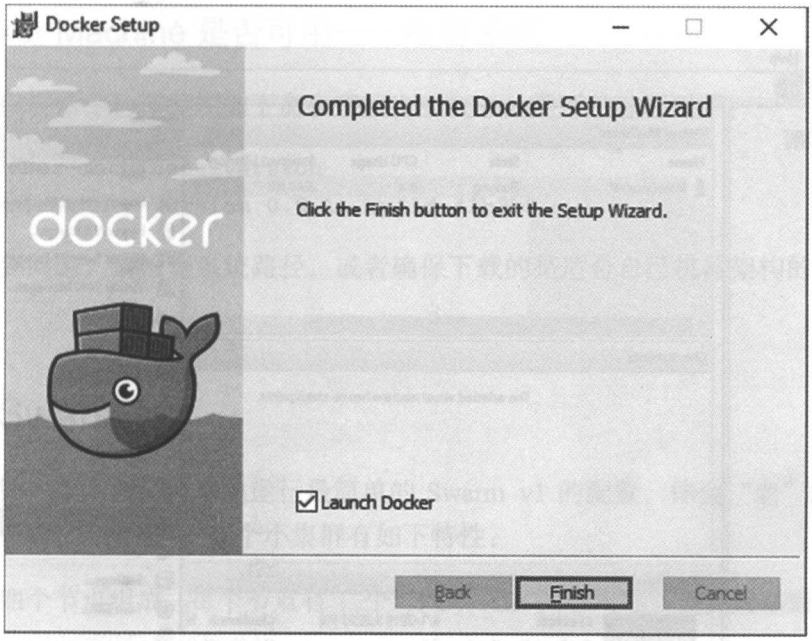


Windows 系统上的 Docker

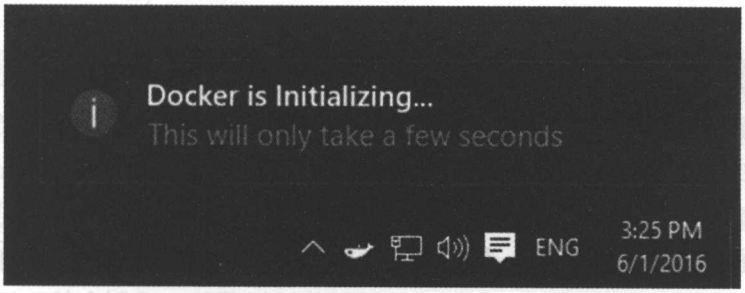
在 Windows 系统上安装 Docker，要求操作系统是启用了 Hyper-V 的 Windows 10。一般来说，Windows 10 专业版或者更高版本都自带 Hyper-V。双击 Setup 程序之后，可以看到第一屏显示 License Agreement，类似下图。安装程序会要求用户输入秘钥，这与 Mac 系统上的 Docker 类似。



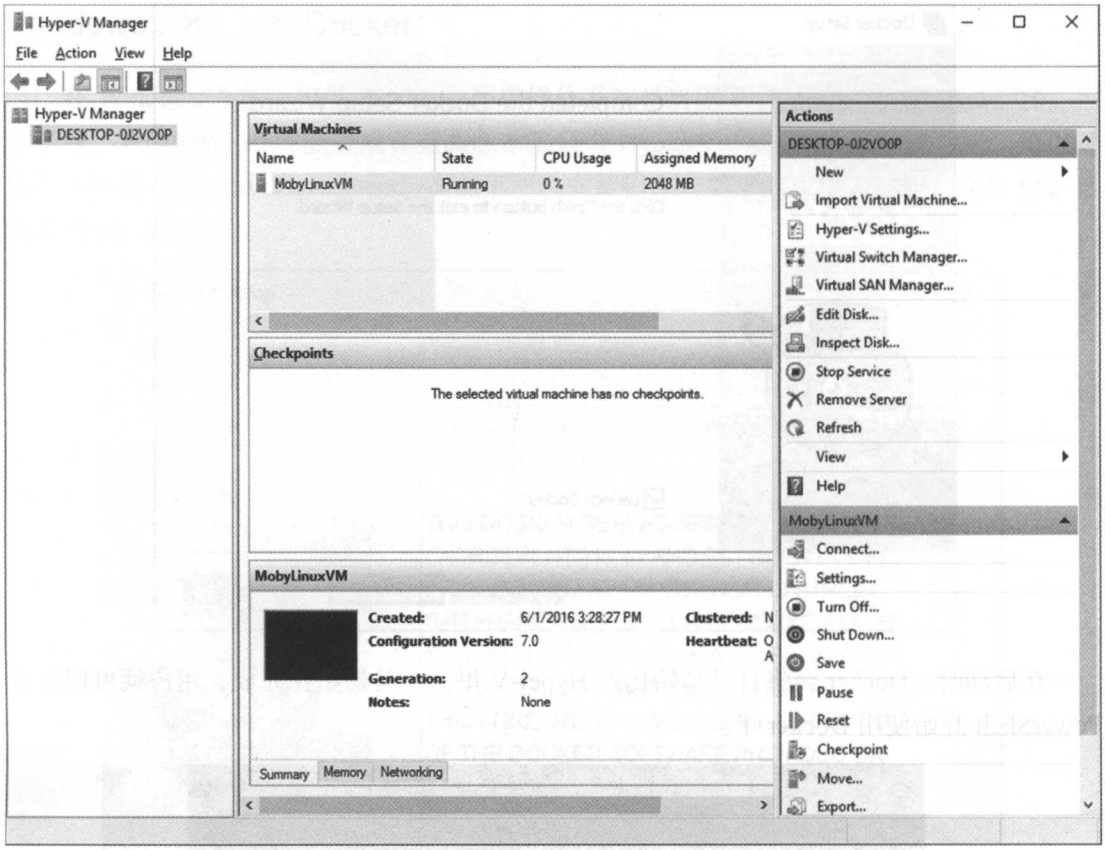
如果安装正常的话，用户会看到结束窗口，提示用户可以启动 Windows 系统上的 Docker，如下图所示。



在启动时，Docker 会将自己初始化到 Hyper-V 里。一旦初始化完成，用户就可以打开 PowerShell 开始使用 Docker 了。



如果哪里出错了，可以从系统托盘图标的菜单里打开日志窗口，也可以检查 Hyper-V 管理器。



使用 Linux

本书会大量使用 Machine，因此请确保已经通过 Mac 或 Windows 系统上的 Docker 或者 Docker Toolbox 安装了 Machine。如果你的电脑使用的是 Linux，那么可以使用包系统（apt 或者 rpm）来安装 Docker 客户端。我们还需要下载纯机器的二进制文件，只需要 curl 命令下载文件并且赋给它执行权限；按照 https://docs.docker.com/machine/install-machine/处的指令即可轻松完成。目前的稳定版本是 0.8.1。

```
$ curl -L
https://github.com/docker/machine/releases/download/v0.8.1/dockermachine-
uname -s-uname -m > /usr/local/bin/docker-machine
$ chmod +x /usr/local/bin/docker-machine
```

检查 Docker Machine 是否可用——所有系统

用户可以在命令行里使用如下命令来检查 Machine 是否已经可用了：

```
$ doc006Ber-machine --version
docker-machine --version 0.8.1, build 41b3b2
```

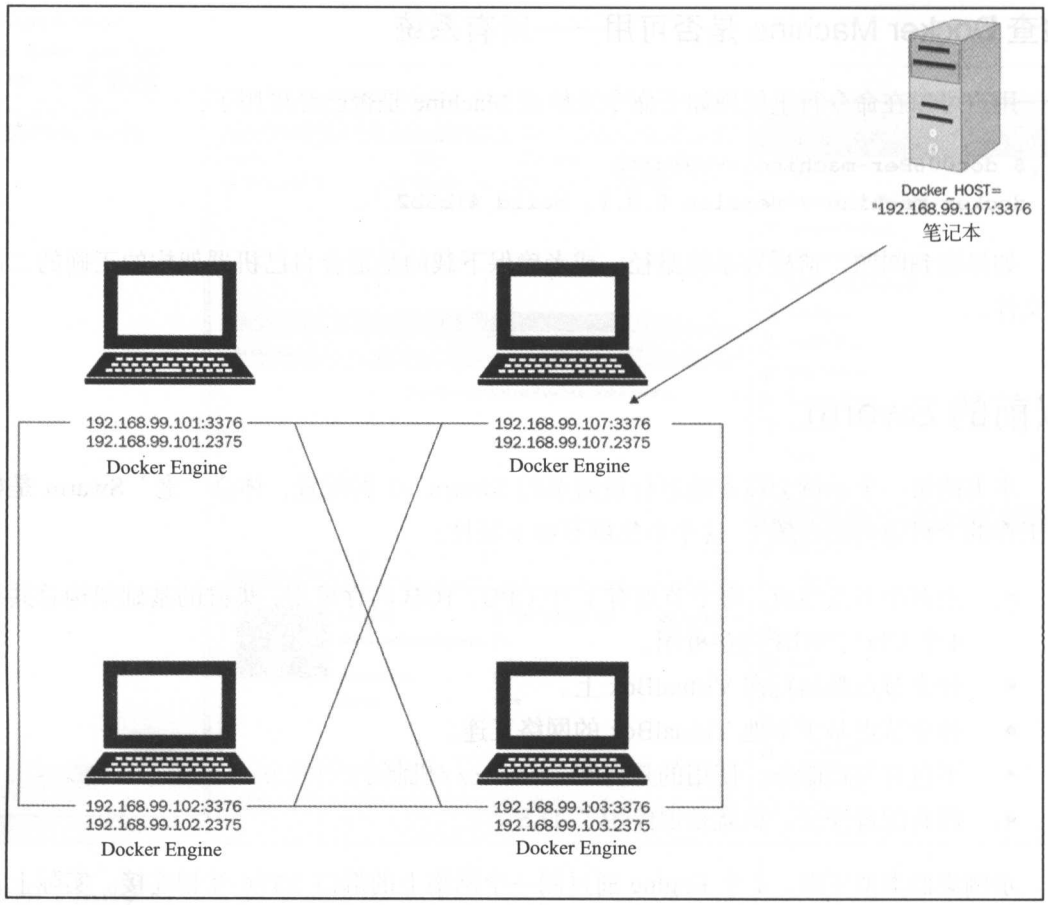
如果遇到问题，请检查系统路径，或者确保下载的是适合自己机器架构的正确的二进制文件。

以前的 Swarm

本书的第一个示例会在本地运行最简单的 Swarm v1 的配置，体会“老”Swarm 是如何工作的（目前仍然有效）。这个小集群有如下特性：

- 由四个节点组成，每个节点有 1 个 CPU，1GB 内存可用；集群的基础架构总共有 4 个 CPU、4GB 内存可用。
- 每个节点都运行在 VirtualBox 上。
- 每个节点基于本地 VirtualBox 的网络互连。
- 不包含发现服务，使用的是静态 `nodes://`机制。
- 没有配置安全，也就是说停用了 TLS。

示例集群类似下图。4 个 Engine 通过同一个网络上的端口 `3376` 互相连接。实际上，除了 Docker Engine，每个节点还会运行一个 Docker 容器，在主机上暴露端口 `3376`（Swarm），并且将其重定位到自身。运维人员通过将环境变量 `DOCKER_HOST` 设置为 `IP:3376`，就能够连接到（任意）主机上。按照下面的示例一步步完成，所有东西都很明了。



首先，需要使用 Docker Machine 创建四个 Docker 主机。Docker Machine 将这些步骤都自动化了，只需要使用一个命令即可，无须自己手动创建一个 Linux VM，生成并且上传证书，通过 SSH 登录，并且安装及配置 Docker daemon。

Machine 会执行如下步骤：

1. 基于 boot2docker 镜像创建一个 VirtualBox VM。
2. 给这个 VM 分配一个 VirtualBox 内部网络上的 IP。
3. 上传并且配置证书和秘钥。
4. 在这个 VM 里安装 Docker daemon。

5. 配置 Docker daemon 并且将其暴露，从而可以远程访问它。

最终，用户将得到一个运行着 Docker 的 VM，并且可以被访问来运行容器。

Boot2Docker

Boot2Docker 使用 Tiny Core Linux 构建，是一个轻量级的发行版，其被特别设计用来运行 Docker 容器。它完全在 RAM 里运行，启动极其迅速，从开始到完成大概需要五秒钟。当启动 Engine 时，Boot2Docker 默认在安全端口 2376 启动 Docker Engine。

Boot2Docker 并不适合生产环境的工作负载。它仅仅是为开发和测试而设计的。本书会从使用 Boot2Docker 开始，然后在后续章节里介绍生产环境。在撰写本书时，Boot2Docker 支持 Docker 1.12.3 版本，并且使用的是 Linux Kernel 4.4。Docker Engine 默认使用的是 AUFS 4 作为存储驱动。

使用 Docker Machine 创建 4 个集群节点

执行如下命令：

```
$ docker-machine ls
```

可以列出可用的机器，我们可以看到还没有运行着的机器。那么，现在就开始创建一个机器吧，使用如下命令：

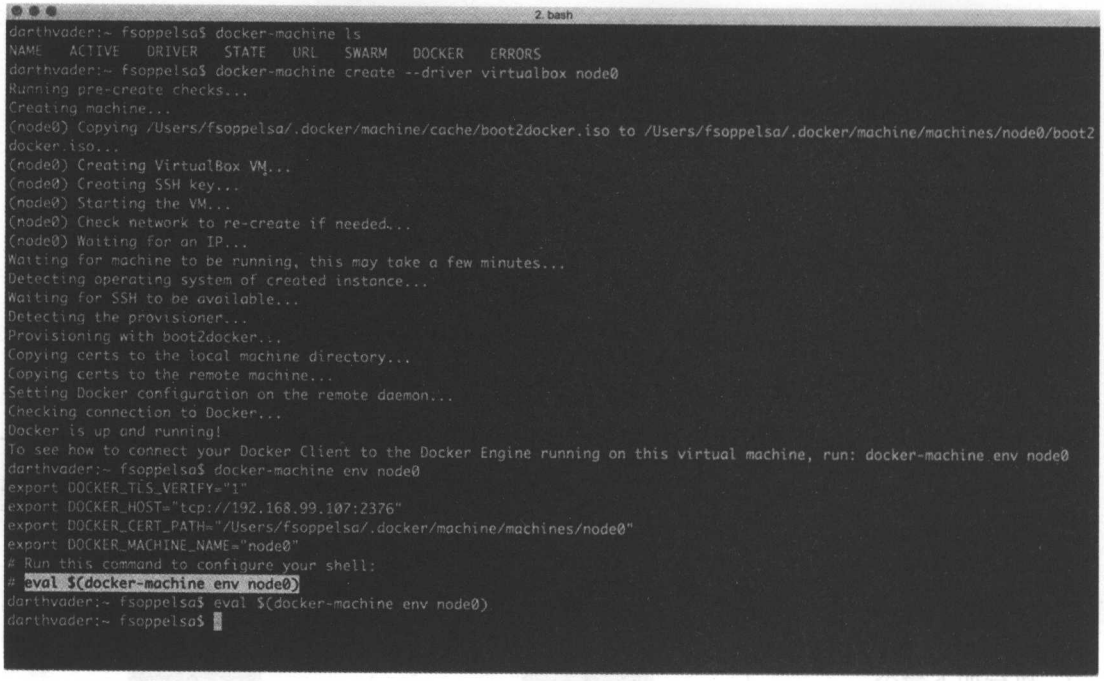
```
$ docker-machine create --driver virtualbox node0
```

该命令指定使用 VirtualBox 驱动（可以简单使用-d 代替）并且将该机器命名为 node0。Docker Machine 能够在很多不同的公有云和私有云供应商上预配机器，比如 AWS、DigitalOcean、Azure、OpenStack，并且提供很多参数。这里使用的是标准设置。一段时间后第一个集群节点就创建好了。

这时，运行如下命令可以访问该主机（远程获得访问权限）：

```
$ docker-machine env node0
```

其会打印出一些 shell 变量。只需要复制最后一行（带有 eval 的那行），粘贴并且输入 Enter。配置了这些变量之后，你操作的就不再是本地 daemon 了（如果本地 daemon 存在的话），而是 `node0` 的 Docker daemon。



再次检查机器列表，我们将会看到镜像名称旁出现了一个*号，表示目前这台机器已经在使用中了。另外，还可以键入如下命令打印出当前活跃的机器：

```
$ docker-machine active
```

```
darthvader:~ fsoppelsa$ docker-machine active
node0
darthvader:~ fsoppelsa$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
node0   *        virtualbox   Running   tcp://192.168.99.107:2376           v1.11.1
```

deamon 运行在这台机器上，使用的是标准设置（比如使用端口 tcp/2376，启用 TLS）。可以通过 SSH 到该节点并验证运行着的进程来确认设置信息：

```
$ docker-machine ssh node0 ps aux | grep docker
1320 root /usr/local/bin/docker daemon -D -g /var/lib/docker -H
unix:// -H tcp://0.0.0.0:2376 --label provider=virtualbox --
tlsverify --tlscacert=/var/lib/boot2docker/ca.pem --
tlscert=/var/lib/boot2docker/server.pem --
tlskey=/var/lib/boot2docker/server-key.pem -s aufs
```

这时我们就可以立刻使用 Docker daemon 来启动容器以及检查 Docker 的状态：

```
2. bash
darthvader:~ fsoppelsa$ docker run -ti busybox sh
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox

385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:4a887a2326ec9e0fa90cce7b4764b0e627b5d6afcb81a3f73c85dc29cea00048
Status: Downloaded newer image for busybox:latest
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
       valid_lft forever preferred_lft forever
/ # darthvader:~ fsoppelsa$ docker info | head -10
Containers: 1
 Running: 0
 Paused: 0
 Stopped: 1
Images: 1
Server Version: 1.11.1
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 4
darthvader:~ fsoppelsa$
```

棒极了！现在以同样的方式继续预配其他三台主机，分别命名为 node1、 node2 和 node3:

```
$ docker-machine create --driver virtualbox node1
$ docker-machine create --driver virtualbox node2
$ docker-machine create --driver virtualbox node3
```

完成后，我们就拥有了四台可用的 Docker 主机。使用 Docker machine 命令验证。

```
darthvader:~ fsoppelsa$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
node0   *        virtualbox   Running   tcp://192.168.99.107:2376           v1.11.1
node1   -        virtualbox   Running   tcp://192.168.99.101:2376           v1.11.1
node2   -        virtualbox   Running   tcp://192.168.99.102:2376           v1.11.1
node3   -        virtualbox   Running   tcp://192.168.99.103:2376           v1.11.1
```

这时就已经可以启动 Swarm 集群了。但是，在这之前，为了保证第一个示例尽可能简单，先让这些运行着的 Engine 停用 TLS。我们的计划是：在端口 2375 运行 Docker daemon，而不使用 TLS。

这里详细介绍一下所有端口的细节信息。

不安全	安　全
Engine：2375	Engine：2376
Swarm：3375	Swarm：3376
	Swarm v2 使用 2377 作节点发现

端口 2377 是 Swarm v2 节点用来发现集群里的其他节点的。

配置 Docker 主机

要理解 TLS 配置在哪里，需要通过关闭所有 Docker 主机的 TLS 来做一些试验。另外这里关闭 TLS 也是想给读者一些启发，swarm manage 命令是如何通过调用自身来工作的。

我们有四台主机，在端口 tcp/2376 运行 Docker，并且启用了 TLS，因为 Docker Machine 默认会创建这些。这里需要重新配置，将 daemon 端口更改为 tls/2375，并且移除 TLS。因此，使用如下命令登录每个节点：

```
$ docker-machine ssh node0
```

然后，获得 root 权限：

```
$ sudo su -
```

还要通过修改文件/var/lib/boot2docker/profile 来配置 boot2docker：

```
# cp /var/lib/boot2docker/profile /var/lib/boot2docker/profile-bak
# vi /var/lib/boot2docker/profile
```

删除包含 CACERT、SERVERKEY 和 SERVERCERT 的行，并且将 daemon 端口配置为 tcp/2375，DOCKER_TLS 配置为 no。实际配置如下：

```
EXTRA_ARGS='
--label provider=virtualbox

'
DOCKER_HOST='-H tcp://0.0.0.0:2375'
DOCKER_STORAGE=aufs
DOCKER_TLS=no
```

之后从 SSH 会话里登出，并且重启机器：

```
$ docker-machine restart node0
```

现在 Docker 就运行在端口 tcp/2375 并且没有安全配置了。可以使用如下命令验证：

```
$ docker-machine ssh node0 ps aux | grep docker
 1127 root /usr/local/bin/docker daemon -D -g /var/lib/docker -H
 unix:// -H tcp://0.0.0.0:2375 --label provider=virtualbox -s aufs
```

最终，在本地电脑里，删除 DOCKER_TLS_VERIFY 的设置，并且重新导出 DOCKER_HOST，从而最终使用没有 TLS 的 daemon 侦听 tcp/2375 端口：

```
$ unset DOCKER_TLS_VERIFY
$ export DOCKER_HOST="tcp://192.168.99.103:2375"
```

在组成第一个 Swarm 的其他三个节点上重复上述操作。

启动 Docker Swarm

要开始使用 Swarm v1，用户必须首先从 Docker hub 里拉取 swarm 镜像。打开四个终端，在每个终端上导入每台机器的环境变量。第一个终端导入 node0（用 docker-machine env node0 命令，并且将变量复制粘贴到 shell 上），第二个终端导入 node1，以此类推。如上文所述，完成了变更标准端口并且停用 TLS 之后，在每台机器上做如下操作：

```
$ docker pull swarm006D
```

```
darthvader:~ fsoppelsa$ docker pull swarm
Using default tag: latest
latest: Pulling from library/swarm

51436fd4bb0d: Pull complete
c31a5390266f: Pull complete
e40019be13ea: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:3add485cb6bb71c7113243753c5f484561549d9f782154b1c809219c9754ce46
Status: Downloaded newer image for swarm:latest
```

第一个示例中不使用发现服务，而是使用最简单的机制，比如 `nodes://`，将 Swarm 集群节点手动连接上，形成机器网格。运维人员仅仅需要简单定义节点 IP 列表以及 daemon 端口，用逗号将 IP 端口隔开，如下所示：

```
nodes://192.168.99.101:2375,192.168.99.102:2375,192.168.99.103:2375,192.168.99.107:2375
```

要使用 Swarm，仅仅需要使用一些参数运行 Swarm 容器。键入如下命令查看在线帮助文档：

```
$ docker run swarm --help
```

```
darthvader:~ fsoppelsa$ docker run swarm --help
Usage: swarm [OPTIONS] COMMAND [arg...]

A Docker-native clustering system

Version: 1.2.3 (eaa53c7)

Options:
  --debug                          debug mode [$DEBUG]
  --log-level, -l "info"           Log level (options: debug, info, warn, error, fatal, panic)
  --experimental                   enable experimental features
  --help, -h                       show help
  --version, -v                    print the version

Commands:
  create, c     Create a cluster
  list, l       List nodes in a cluster
  manage, m     Manage a docker cluster
  join, j       Join a docker cluster
  help          Shows a list of commands or help for one command

Run 'swarm COMMAND --help' for more information on a command.
```

如上所见，Swarm 有四大基本命令：

- **Create** 用来创建带有发现服务的集群，比如 `token://`。
- **List** 展示集群节点列表。
- **Manage** 允许用户操作某个 Swarm 集群。
- **Join**，和发现服务一起，用来将新节点加入到某个已有集群里。

这里将使用 `manage` 命令。这是参数最多的命令（可以通过运行 `docker run swarm manage --help` 查看所有参数）。本书这里仅仅用来连接节点。如下是每个节点上的策略：

1. 通过 Swarm 容器暴露 Swarm 服务。

2. 以 `daemon (-d)` 模式运行该容器。

3. 将标准的 Swarm 端口 `tcp/3376` 转发到内部（容器内）端口 `tcp/2375`。

4. 使用 `nodes://` 指定主机列表为集群的一部分——每个主机都必须是 `IP:port` 对，这里的端口就是 Docker Engine 的端口（`tcp/2375`）。

让之前的每个终端都连接到每台机器上，执行如下命令：

```
$ docker run \
-d \
-p 3376:2375 \
swarm manage \
nodes://192.168.99.101:2375,192.168.99.102:2375,
192.168.99.103:2375,192.168.99.107:2375
```

当使用 `nodes://` 机制时，用户可以使用类似 Ansible 的主机范围模式，因此可以将如下三个连续的 IP——`nodes://192.168.99.101:2375,192.168.99.102:2375` 和 `192.168.99.103:2375` 的语法简化为 `nodes://192.168.99.[101:103]:2375`。

下一步，在开始使用运行着的容器之前，先连接进去并且查看其信息。为了方便，开启一个新的终端。这里不再连接到某个节点里的 Docker Engine 上，而是连接到 Docker Swarm。因此连接到 `tcp/3376`，而不是 `tcp/2375`。为了展示实验的细节，首先导入 `node0` 的变量：

```
$ docker-machine env node0
```

如上文所述，复制粘贴 eval 行，使用如下命令检查已经导入了哪些 shell 变量：

```
$ export | grep DOCKER_
```

现在需要完成如下任务：

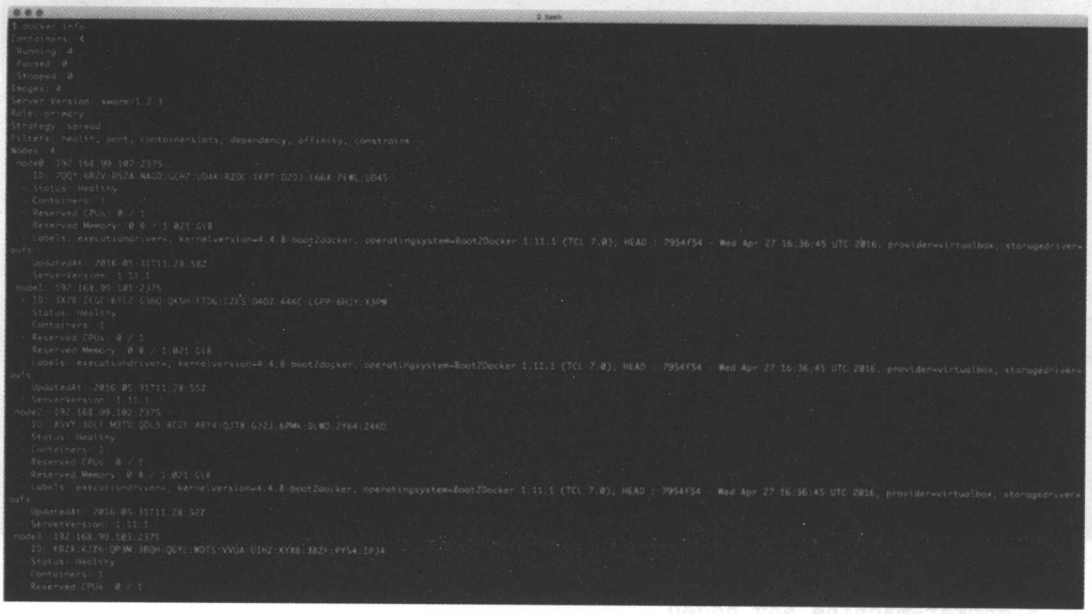
1. 改变 DOCKER_HOST，连接到 Swarm 端口 tcp/3376，而不是 Engine 端口 tcp/2375。
2. 停用 DOCKER_TLS_VERIFY。
3. 停用 DOCKER_CERT_PATH。

```
$ export DOCKER_HOST="tcp://192.168.99.107:3376"
$ unset DOCKER_TLS_VERIFY
$ unset DOCKER_CERT_PATH
```

配置应该如下所示：

```
$ export | grep DOCKER
declare -x DOCKER_HOST="tcp://192.168.99.107:3376"
declare -x DOCKER_MACHINE_NAME="node0"
```

现在如果在端口 3376 连接 Docker Swarm，并且展示信息，我们可以看到 Swarm 已经正在运行中了：



恭喜恭喜！你刚刚已经使用 Swarm 启动了第一个 Docker 集群。可以看到除了四个 Swarm 外，集群里还没有其他运行的容器，但是 Server Version 是 Swarm/1.2.3，调度策略已经分发，并且最为重要的是，在 Swarm 里有四个健康的节点（之后显示了每个 Swarm 节点的细节）。

另外，我们可以得到该 Swarm 集群的调度行为相关的额外信息：

```
Strategy: spread
Filters: health, port, containerslots, dependency, affinity,
constraint
```

spread 调度策略意味着 Swarm 将会尝试在资源利用少的主机上放置容器，并且在创建容器时可以使用列出的过滤器，从而允许用户手动输入一些参数。比如，用户可能想要让 Galera 集群容器物理上接近，但是在不同的主机上。

不过，这个 Swarm 的大小是多少呢？我们可以在输出的最后看到：

```
Kernel Version: 4.4.8-boot2docker
Operating System: linux
Architecture: amd64
CPUs: 4
Total Memory: 4.085 GiB
```

也就是说这个迷你 Swarm 总共拥有这些资源：四个 CPU 以及 4GB 内存。这和预期完全一致，四个 VirtualBox 主机，每个 VirtuaBox 主机有一个 CPU 和 1GB 内存。

测试 Swarm 集群

至此，我们已经拥有了一个 Swarm 集群，现在就可以开始使用它了。这里会演示 spread 策略算法，容器将放置到负载较少的主机上。本例非常简单，从四个空白的节点开始。首先，连接到 Swarm，Swarm 将容器放置到主机上。启动一个 ngnix 容器，将其端口 `tcp/80` 映射到主机（机器）端口 `tcp/80` 上。

```
$ docker run -d -p 80:80 nginx
2c049db55f9b093d19d575704c28ff57c4a7a1fb1937bd1c20a40cb538d7b75c
```

可以看到 Swarm 调度器决定将这个容器放到 `node1` 上：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 STATUS              PORTS                                NAMES
2c049db55f9b        nginx               "nginx -g 'daemon off"   About a minute ago
 Up About a minute   192.168.99.101:80->80/tcp, 443/tcp   node1/dreamy_goldberg
```

因为需要将端口 tcp/80 绑定到所用主机上，我们仅仅有四次机会将四个容器放到四个不同的主机上。创建一个新的 nginx 容器，看看会发生什么情况：

```
$ docker run -d -p 80:80 nginx
577b06d592196c34ebff76072642135266f773010402ad3c1c724a0908a6997f
$ docker run -d -p 80:80 nginx
9fabe94b05f59d01dd1b6b417f48155fc2aab66d278a722855d3facc5fd7f831
$ docker run -d -p 80:80 nginx
38b44d8df70f4375eb6b76a37096f207986f325cc7a4577109ed59a771e6a66d
```

我们可以看到四个 nginx 容器放到了四台 Swarm 主机上：

```
$ docker ps --format "{{.ID}}: {{.Ports}} {{.Command}}"
38b44d8df70f: 192.168.99.103:80->80/tcp, 443/tcp "nginx -g 'daemon off"
9fabe94b05f5: 192.168.99.102:80->80/tcp, 443/tcp "nginx -g 'daemon off"
577b06d59219: 192.168.99.107:80->80/tcp, 443/tcp "nginx -g 'daemon off"
2c049db55f9b: 192.168.99.101:80->80/tcp, 443/tcp "nginx -g 'daemon off"
```

然后尝试创建一个新的 nginx：

```
$ docker run -d -p 80:80 nginx
docker: Error response from daemon: Unable to find a node that satisfies the
following conditions
[port 80 (Bridge mode)].
See 'docker run --help'.
```

Swarm 找不到合适的主机放置这个新容器，因为所有主机上的端口 tcp/80 都已经被占用了。运行这四个 ngnix 容器以及四个 Swarm 容器（用于基础架构的管理）之后，这个 Swarm 集群里一共有八个运行着的容器：

```
$ docker info
Containers: 8
 Running: 8
 Paused: 0
 Stopped: 0
```

这也正是 Swarm v1 的工作方式。

如今的 Swarm

本节使用 Docker Engine 1.12 版本或更新版本所内建的 Swarm Mode 搭建一个小型的集群。

在 DockerCon16 所有的消息里，容器编排相关的两个消息最引人注目：

- Engine 和 Swarm 的集成，称为 Docker Swarm Mode。
- SwarmKit

实际上，Docker daemon 从 1.12 版本开始，就已经可以运行所谓的 Swarm Mode。Docker 客户端里添加了新的 CLI 命令，比如 `Node`、`Service`、`Stack`、`Deploy`，当然还有 `Swarm`。

本书从第 3 章“遇见 Docker Swarm Mode”开始，会介绍更多 Swarm Mode 和 SwarmKit 的知识，但是既然本章已经使用 Swarm v1 完成了前面的示例，那么这里也用 Swarm v2 让读者先体会一下 v2 和 v1 相比更为简单的用户体验。使用 Swarm v2 的唯一要求是 daemon 的版本至少是 1.12-rcl。但是只要使用的是 Docker Machine 0.8.0-rcl+版本，就可以正常预配出满足需求的 Docker 主机。

DockerCon 2016 上，Docker 还发布了在 AWS 上和在 Azure 上使用的 Docker。市场里不仅仅有 AWS 和 Azure，实际上我们还喜欢使用 DigitalOcean，因此我们自己创建了一个新工具，封装了 `doctl`，DigitalOcean 的命令行接口可以以全新的大批量的方式帮助预配 Docker 集群。工具的名字为 `belt`，可以在 Github 的 http://github.com/chanwit/belt 仓库获取到。可以使用如下命令获得 belt：

```
go get github.com/chanwit/belt
```

或者从该项目的 Release 页面里下载二进制文件。

首先，为 DigitalOcean 的预配准备模板文件。`.belt.yaml` 类似如下：

```
$ cat .belt.yaml
---
digitalocean:
region: sgp1
image: 18153887
```

```
ssh_user: root
ssh_key_fingerprint: 816630
```

请注意镜像号 18153887 是包含 Docker 1.12 版本的快照。DigitalOcean 通常都使用最新发布的 Docker 镜像。要能够控制集群的话，还需要 SSH 密钥。在字段 ssh_key_fingerprint 里，可以输入 finger print 或者密钥 ID。

不要忘记设置 DIGITALOCEAN_ACCESS_TOKEN 环境变量。另外，Belt 能够识别相同的 Docker Machine shell 变量。如果你熟悉 Docker Machine 的话就应该知道如何设置它们。如下是前一节设置的 shell 变量：

- export DOCKER_TLS_VERIFY="1"
- export DOCKER_HOST="tcp://<IP ADDRESS>:2376"
- export DOCKER_CERT_PATH="/Users/user/.docker/machine/machines/machine"
- export DOCKER_MACHINE_NAME="machine"

现在看看如何使用 Belt：

```
$ export DIGITALOCEAN_ACCESS_TOKEN=1b207 .. snip .. b6581c
```

创建四个节点的 Swarm，每个节点有 512MB 内存：

```
$ belt       create  512mb node[1:4]
ID Name      Public  IPv4    Memory  VCPUs   Disk
18511682     node1           512     1       20
18511683     node4           512     1       20
18511684     node3           512     1       20
18511681     node2           512     1       20
```

如上可见，我们可以使用 node[1:4]这样简单的语法来指定一系列节点。该命令会在 DigitalOcean 上创建四个节点。大概需要等待 55 秒就可以预配完所有节点。然后就可以查看了：

```
$ belt ls
ID          Name    Public IPv4         Status Tags
18511681    node2   128.199.105.119     active
18511682    node1   188.166.183.86      active
18511683    node4   188.166.183.103     active
```

```
18511684    node3   188.166.183.157    active
```

它们的状态已经从“new”变更为“active”。所有的 IP 地址都已经分配好了。一切看起来都很顺利。

现在启动 Swarm。

在这之前确保运行的是 Docker 1.12 版本。在 node1 上验证这一点。

```
$ belt active node1
node1
$ belt docker version
Client:
 Version: 1.12.0-rc2
 API version: 1.24
 Go version: go1.6.2
 Git commit: 906eacd
 Built: Fri Jun 17 21:02:41 2016
 OS/Arch: linux/amd64
 Experimental: true
Server:
 Version: 1.12.0-rc2
 API version: 1.24
 Go version: go1.6.2
 Git commit: 906eacd
 Built: Fri Jun 17 21:02:41 2016
 OS/Arch: linux/amd64
 Experimental: true
```

belt docker 命令仅仅是一个简单的封装器命令，将整个命令行通过 SSH 发送到 Docker 主机里。因此这个工具不会成为问题，用户可以一直控制 Docker Engine。

这里使用 Swarm Mode 初始化第一个节点。

```
$ belt docker swarm init
Swarm initialized: current node (c0llmsc5t1tsbtcblrx6ji1ty) is now
a manager.
```

接下来让其他三个节点也加入这个新建的集群。组建大型集群是项很枯燥的任务。不用用户遍历每个节点和手动完成 docker swarm join 的操作，belt 可以帮助用户自动完成：

```
$ belt swarm join node1 node[2:4]
node3: This node joined a Swarm as a worker.
node2: This node joined a Swarm as a worker.
node4: This node joined a Swarm as a worker.
```

当然我们也可以运行：belt --host node2 docker swarm join <node1's IP>:2377 手动将 node2 加入到集群里。

这时集群信息如下：

```
$ belt docker node ls
ID            NAME MEMBERSHIP  STATUS  AVAILABILITY MANAGER STATUS
4m5479vud9qc6qs7wuy3krr4u   node2   Accepted    Ready   Active
4mkw7ccwep8pez1jfeok6su2o   node4   Accepted    Ready   Active
a395rnht2p754w1beh74bf7f1   node3   Accepted    Ready   Active
c0llmsc5t1tsbtcblrx6ji1ty   * node1 Accepted    Ready   Active
Leader
```

祝贺！至此，已经在 DigitalOcean 上安装好一个 Swarm 集群了。

现在为 nginx 创建一个服务。如下命令可以创建出一个 Nginx 服务，包含 2 个容器实例，使用端口 80。

```
$ belt docker service create --name nginx --replicas 2 -p 80:80
nginx
d5qmntf1tvvztw9r9bhx1hokd
```

查看服务信息：

```
$ belt docker service ls
ID NAME REPLICAS IMAGE COMMAND
d5qmntf1tvvz nginx 2/2 nginx
```

将其扩展到 4 个节点。

```
$ belt docker service scale nginx=4
```

```
nginx scaled to 4
$ belt docker service ls
ID NAME REPLICAS IMAGE COMMAND
d5qmntf1tvvz nginx 4/4 nginx
```

和 Docker Swarm 类似，可以使用 `belt ip` 来查看节点运行的位置。可以使用任意 IP 地址浏览 NGINX 服务。它在所有节点上都可用。

```
$ belt ip node2
128.199.105.119
```

这就是 Docker 1.12 版本开始提供的 Swarm Mode 功能。

本章小结

本章介绍了 Docker Swarm，定义了其目标、特性和架构。还讨论了 Swarm 的其他可能的开源替代方案以及之间的关系。最后，安装并且开始使用 Swarm，创建了一个简单的本地集群，集群由 VirtualBox 和 Digital Ocean 上的四台主机组成。

使用 Swarm 集群化容器是本书的主要内容，但是在介绍生产环境里的 Swarm 之前，本书会先探讨一些理论知识，从发现服务开始，这是第 2 章“探索发现服务”的主要内容。

第 2 章

探索发现服务

第 1 章“欢迎来到 Docker Swarm”里创建了一个简单但是工作良好的本地 Docker Swarm 集群，使用的是 `nodes://`发现机制。这个系统并不很实用，只能用来学习 Swarm 的基础功能。

实际上，它仅仅是一个简单模型，并没有真正使用任何 master-slave 架构，更不用说更高层级的服务，比如节点发现和自动配置、弹性、选主以及故障恢复（高可用）。因此，它并不适用于生产环境。

除了 `nodes://`发现机制，Swarm v1 还正式支持四种发现服务。不过，其中的 Token 服务仅适用于非生产环境。使用 Swarm v1，用户仍然需要手动集成一种发现服务，而使用 Swarm Mode（Docker 1.12 版本中），则已经集成了一种发现服务——Etcd。本章会介绍：

- 发现服务
- 测试用的发现服务：Token
- Raft 理论和 Etcd
- ZooKeeper 和 Consul

在深入研究这些服务之前，我们先一起思考一下什么是发现服务？

发现服务

假定在静态配置上运行 Swarm 集群，这样的配置和第 1 章“欢迎来到 Docker Swarm”里的示例类似。这时，网络是 flat 的，并且每个容器都指派了一个特定的任务，比如 MySQL 数据库。定位 MySQL 容器很容易，因为系统给这个容器分配了一个定义好的 IP 地址，或者系统里运行着某种 DNS 服务器。了解这个容器的工作状态也很简单，因为它不会变更其端口（`tcp/3336`）。并且 MySQL 容器也并不需要向外声明自身是可用的数据库容器，以及发布自己的 IP 和端口，因为用户已经知道这些信息了。

这正是宠物模型的体现，系统由系统管理员手动维护。但是，我们想成为更加先进的运维人员，想要使用的是牛群模型。

这里设想一下正在运行的是一个 Swarm 集群，由数百个节点组成，托管了多个应用程序，运行一定数量的服务（Web 服务器、数据库、键值存储、缓存和队列）。这些应用程序运行在大量容器上，它们可能会动态地改变其 IP 地址，要么是因为重启，创建了新容器，启动了副本，要么是因为某些高可用机制启动了一些新容器。

那么，怎么才能找到支撑 Acme 应用程序的 MySQL 服务呢？如何确保负载均衡器能够知道 100 个 Nginx 前端的地址，从而保证功能的正常运行呢？如果某个服务发生了变更，使用了和以前不同的配置，怎么才能将这样的变动通知到所有相关方呢？

答案是使用发现服务。

所谓的发现服务是一种包含很多特性的机制。可选的服务很多，功能类似，各有优劣，但是基本上，所有的发现服务都是针对分布式系统的，因此它们必须分布在所有集群节点上，能够扩展以及容错。发现服务的主要目标是帮助服务找到其他服务并且互相通信。为了达成这一目标，它们需要通过发布自身信息来保存（注册）相关信息，比如每个服务的位置在哪里等，这些信息通常以键值存储的形式存在。发现服务在 Docker 出现之前就已经存在了，但是随着容器和容器编排的兴起，这个服务才变得越来越重要。

总之，通过发现服务：

- 可以定位基础架构上的单个服务
- 可以通知服务配置的变更

- 服务可以注册其可用性
- 等等

发现服务通常是键值存储。Docker Swarm v1 正式支持如下发现服务。但是，用户也可以使用 `libkv` 抽象接口集成自己的发现服务，示例如下：https ://github.com/docker/docker/tree/ master/pkg/discovery。

- Token
- Consul 0.5.1+
- Etcd 2.0+
- ZooKeeper 3.4.5+

注意，Swarm mode 已经集成了 Etcd 库，作为内建的发现服务。

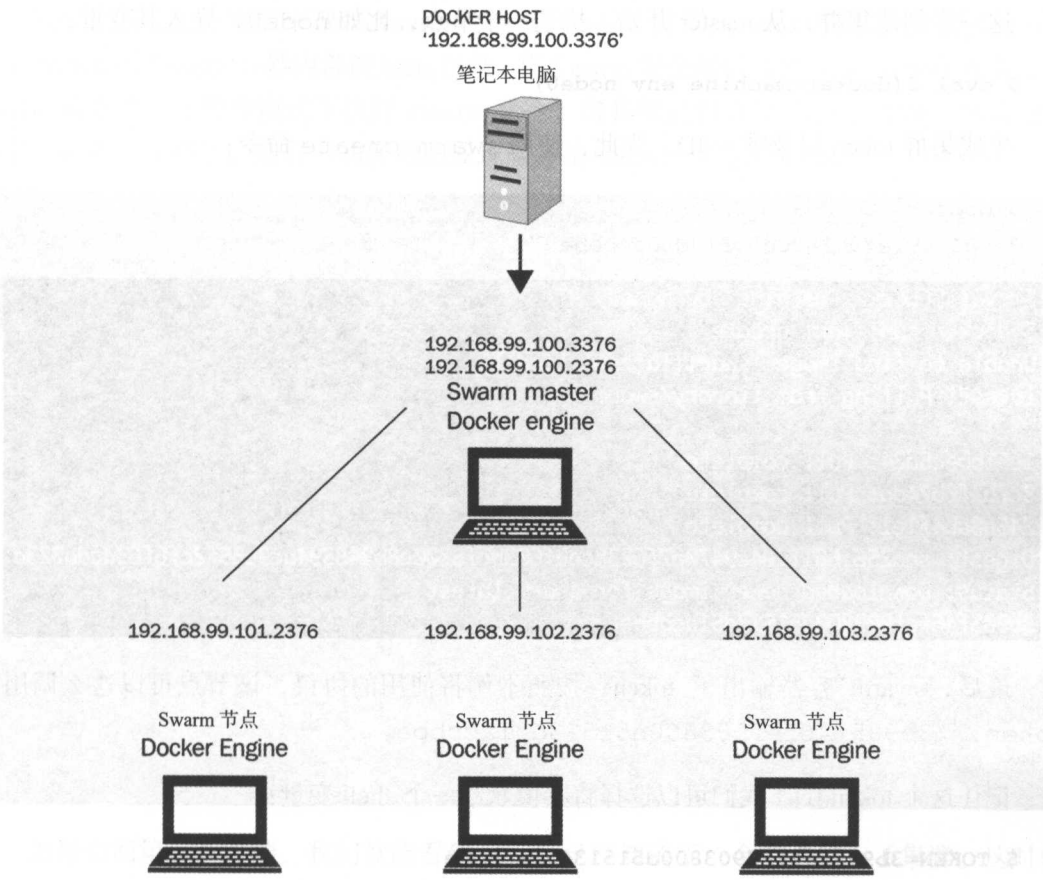
Token

Docker Swarm v1 包含一种开箱即用的发现服务，称为 Token。Token 集成进了 Docker Hub；因此，它要求所有 Swarm 节点都能够连网并且能够访问 Docker Hub。这是 Token 的主要限制所在，不过我们在下文很快就能够看到，Token 能够帮助我们练习如何处理集群。

使用 token 重新架构第 1 章示例

如果想要练习实际操作的能力，现在是时候研究一个实例了。这里使用 token 重新架构第 1 章“欢迎来到 Docker Swarm”的示例。集群实际不应该是 flat 的，它包含一个 master 和三个 slave，并且每个节点默认启用了安全。

master 节点是暴露 Swarm 端口 `3376` 的节点。我们将连接上它从而操作整个集群。



我们可以使用如下命令创建 4 个节点：

```
$ for i in `seq 0 3`; do docker-machine create -d virtualbox
node$i;
done
```

现在有了运行着最新版 Engine 的四台机器，并且启用了 TLS。这意味着，Engine 暴露的端口是 2376，而不是 2375。

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
node0   -        virtualbox   Running   tcp://192.168.99.100:2376           v1.11.2
node1   -        virtualbox   Running   tcp://192.168.99.101:2376           v1.11.2
node2   -        virtualbox   Running   tcp://192.168.99.102:2376           v1.11.2
node3   -        virtualbox   Running   tcp://192.168.99.103:2376           v1.11.2
```

这一步创建集群，从 master 开始。挑选一个节点，比如 node0，导入其变量：

```
$ eval $(docker-machine env node0)
```

生成集群 token 以及唯一 ID。为此，使用 swarm create 命令：

```
$ docker run swarm create
3b905f46fef903800d51513d51acbbbe
```

```
$ eval $(docker-machine env node0)
$ docker run swarm create
Unable to find image 'swarm:latest' locally
latest: Pulling from library/swarm

1e61bbec5d24: Pull complete
8c7b2f6b74da: Pull complete
245a8db4f1e1: Pull complete
Digest: sha256:661f2e4c9470e7f6238cebf603bcf5700c8b948894ac9e35f2cf6f63dcda723a
Status: Downloaded newer image for swarm:latest
3b905f46fef903800d51513d51acbbbe
```

最后，swarm 容器输出了 token，按照本例将使用的协议，该节点可以这么调用：token://3b905f46fef903800d51513d51acbbbe。

记住这个 token ID，我们可以选择将其值赋给一个 shell 变量：

```
$ TOKEN=3b905f46fef903800d51513d51acbbbe
```

现在创建一个 master，并且尝试满足一些基本的标准安全需求，本章将启用 TLS 加密。下文我们可以看到，swarm 命令接受 TLS 参数。但是如何向容器传递密钥和证书呢？本例使用 Docker Machine 生成的证书，其位于主机的/var/lib/boot2docker 处。

我们实际可以将 Docker 主机上的卷 mount 到运行在该 Docker 主机中的容器里。所有这一切都可以远程控制。

在导入了 node0 变量的情况下，使用如下命令启动 Swarm master：

```
$ docker run -ti -v /var/lib/boot2docker:/certs -p 3376:3376 swarm
manage -H 0.0.0.0:3376 -tls --tlscacert=/certs/ca.pem --
tlscert=/certs/server.pem --tlskey=/certs/server-key.pem
token://$TOKEN
```

以交互模式运行容器来观察 Swarm 的输出。然后，将节点/var/lib/boot2docker 目录 mount 到 Swarm 容器内部的/certs 目录。将 Swarm 安全端口 3376 从 node0 重定位到 Swarm 容器里。在管理模式下执行 swarm 命令，将其绑定到 0.0.0.0:3376。然后指定证书参数和文件路径，最后声明使用的发现服务是 token，并传入 token。

```
$ eval $(docker-machine env node0)
$ docker run -ti -v /var/lib/boot2docker:/certs -p 3376:3376 swarm manage -H 0.0.0.0:3376 -tls --tlscacert=/certs/
ca.pem --tlscert=/certs/server.pem --tlskey=/certs/server-key.pem token://$TOKEN
    [0000] Listening for HTTP                          =0.0.0.0:3376      =tcp
    [0001] Registered Engine node0 at 192.168.99.100:2376
```

保持该节点的运行，开启另一个终端，将第二个节点加入 Swarm 里。首先导入 node1 的变量。Swarm 需要使用 join 命令，从而让该节点加入到 master 为 node0 的集群里：

```
$ docker run -d swarm join --addr=192.168.99.101:2376
token://$TOKEN
```

这里将指定地址为 192.168.99.101 的主机（自身）加入到集群里。

```
$ eval $(docker-machine env node1)
$ export | grep DOCKER_HOST
declare -x DOCKER_HOST="tcp://192.168.99.101:2376"
$ docker run -ti swarm join --addr=192.168.99.101:2376 token://$TOKEN
    [0000] Registering on the discovery service every 1m0s...      =192.168.99.101:2376          =token://3b905f46
fef903800d51513d51acbbbe
```

如果切回第一个终端，我们就能看到 master 已经知道了一个节点加入了集群。这时就已经拥有了一个 Swarm 集群，由一个 master 和一个 slave 组成。

```
[0180] Registered Engine node1 at 192.168.99.101:2376
```

至此，我们已经理解了整个机制，可以在终端停止 docker 命令，并且以-d 参数重新运行。如下，以 daemon 模式运行容器：

```
Master:
$ docker run -t-d -v /var/lib/boot2docker:/certs -p 3376:3376 swarm
manage -H 0.0.0.0:3376 -tls --tlscacert=/certs/ca.pem --
tlscert=/certs/server.pem --tlskey=/certs/server-key.pem
token://$TOKEN
Node:
$ docker run -d swarm join --addr=192.168.99.101:2376
```

```
token://$TOKEN
```

继续把其他两个节点也加入到集群里，导入其变量，并且重复上述命令：

```
$ eval $(docker-machine env node2)
$ docker run -d swarm join --addr=192.168.99.102:2376
token://$TOKEN
$ eval $(docker-machine env node3)
$ docker run -d swarm join --addr=192.168.99.103:2376
token://$TOKEN
```

开启第三个终端，导入 node0 变量，连接到端口 3376（Swarm），而不是 2376（Docker Engine），可以看到 `docker info` 命令的输出如下，集群里有三个节点：

```
$ docker info | egrep "^Node|node"
Nodes: 3
 node1: 192.168.99.101:2376
 node2: 192.168.99.102:2376
 node3: 192.168.99.103:2376
WARNING: No kernel memory limit support
$
```

至此，我们已经成功创建出了一个集群，集群里有一个 master，三个 slave，都启用了 TLS，并且已经可以接收容器了。

在 master 上可以确认该集群的信息，列出集群里的节点。使用 `swarm list` 命令：

```
$ docker run swarm list token://$TOKEN
```

```
$ docker run swarm list token://$TOKEN
192.168.99.103:2376
192.168.99.102:2376
192.168.99.101:2376
```

Token 的限制

Token 虽然还没有被弃用，但是可能很快就会被弃用了。Swarm 里对每个节点的标准需求是要有互联网连接，这很不方便。另外，需要访问 Docker Hub 也让这项技术严重依赖于 Hub 的可用性。实际上，这时 Hub 就会成为单点故障点。但是，使用 token 可以帮助用户理解后台所使用的技术，练习 Swarm v1 的命令：`create`、`manage`、`join` 和 `list`。

至此，我们是时候更进一步，一起了解真正的发现服务和共识算法，这是容错系统里的基本准则。

Raft

共识是分布式系统里的一种算法，强制系统里的代理一致同意某些值，并且选择一个领导者。

众所周知的共识算法是 Paxos 和 Raft。Paxos 和 Raft 性能类似，但是 Raft 相对更为简单，更容易理解，因此在分布式存储实现里更为流行。

作为共识算法，Consul 和 Etcd 实现了 Raft，而 ZooKeeper 实现的是 Paxos。CoreOS Etcd Go 库实现了 Raft，作为依赖条件（在 `vendor/` 目录里）包含在 SwarmKit 和 Swarm Mode 里，因此本书会重点介绍 Raft。

Ongaro 和 Ousterhout 的论文详细介绍了 Raft，该论文位于 https ://ramcloud.stanford.edu/raft.pdf。接下来的内容里会总结其基本观点。

Raft 理论

Raft 的设计目标是简洁，和 Paxos 相比，它也确实实现了这个目标（甚至有学术论文论证这一点）。Raft 和 Paxos 的主要区别在于：在 Raft 里，仅仅由集群领导者向其他节点发送消息和日志，这让该算法更容易理解和实现。在本节里，我们将使用的示例库是由 CoreOS Etcd 实现的 Go 库，位于 https://github.com/coreos/etcd/tree/master/raft。

Raft 集群由节点组成，这些节点必须以一致的方式维护一个副本状态机，集群能够支持新节点的加入、老节点的死机或者不可用，但是这个状态机必须是同步的。

要实现这样的感知故障的目标，Raft 集群通常由奇数个节点组成，比如 3 个或者 5 个节点，来防止脑裂问题。当剩余节点分成几个组，无法在领导者选择上达成一致时，脑裂问题就发生了。如果节点数为奇数，那么它们最终总能通过多数意见选择出领导者。而如果是偶数个节点，选举的最终结果则可能为 50%-50%，而这样的情况是不应该发生的。

回到 Raft 上，Raft 集群在 `raft.go` 里定义为类型为 raft 的结构体，其包含如下信息，比如领导者 UUID、当前 term、日志指针，以及检查 quorum 和选举状态的工具。这里通过分解集群组件和 Node 的定义一步步学习所有概念。Node 在 `node.go` 中被定义为一个接口，在库里实现为 `type node struct`。

```
type Node interface {
    Tick()
    Campaign(ctx context.Context) error
    Propose(ctx context.Context, data []byte) error
    ProposeConfChange(ctx context.Context, cc pb.ConfChange) error
    Step(ctx context.Context, msg pb.Message) error
    Ready() <-chan Ready
    Advance()
    ApplyConfChange(cc pb.ConfChange) *pb.ConfState
    Status() Status
    ReportUnreachable(id uint64)
    ReportSnapshot(id uint64, status SnapshotStatus)
    Stop()
}
```

每个节点是当前运行着的一个实例，都带有一个 tick（通过 `Tick()` 递增），表示任意长度的 term 或者时间周期或者 epoch。在每个 term 里，节点可以处于如下的 StateType：

- Leader（领导者）
- Candidate（候选者）
- Follower（跟随者）

常规情况下，仅存在一个领导者，其他所有节点则都是跟随者。领导者为了让其他人跟随它，会定时向其跟随者发送心跳消息。当跟随者发现不再接收到心跳消息时，它们就知道这个领导者不再可用了，它们随后递增各自的值，成为候选者，并且运行 `Campaign()` 试图成为领导者。它们开始给自己投票，试图达到 quorum 值。当某个节点达到 quorum 值

时，就选举产生了新的领导者。

`Propose()`是提案方法，向日志里增加数据。日志是 Raft 使用的一种数据结构，用来同步集群状态，在 Etcd 里则是另外一个重要概念。它保存在稳定存储（内存）里，当日志变得太大时能够压缩日志从而节约空间（快照）。领导者确保日志永远处在一致的状态，并且仅仅在其确认某个信息已经复制到多数跟随者里因而已经达成一致时，才会将新数据提交到其日志（master-log）里。`Step()`方法用来将状态机推进到下一步。

`ProposeConfChange()`方法用来在运行时变更集群配置。实践已经证明了该方法在任何情况下都是安全的，这归功于两阶段机制，其能够确保该变更得到多数节点的同意。`ApplyConfChange()`方法则负责将该变更应用到当前节点上。

随后是`Ready()`方法。在 Node 接口里，该方法返回一个只读 channel，返回封装好的消息，可以用来读取和保存到存储里，并且提交。通常在调用 Ready 并且应用其实体之后，客户端必须调用`Advance()`，通知大家已经有了一个 Ready 的进程。实际上，`Ready()`和`Advance()`是 Raft 用来保持高层级一致性（coherency）的方法的一部分，避免日志、内容以及状态同步上的不一致。

这正是 CoreOS 的 Etcd 里 Raft 的实现方式。

实际的 Raft

如果你想实际实践一下 Raft，那么可以使用 Etcd 的`raftexample`来启动一个三节点集群。

Docker Compose 的 YAML 文件是自描述的，如下是可以运行的 compose 文件示例：

```
version: '2'
services:
    raftexample1:
        image: fsoppelsa/raftexample
        command: --id 1 --cluster
        http://127.0.0.1:9021,http://127.0.0.1:9022,
        http://127.0.0.1:9023 --port 9121
        ports:
            - "9021:9021"
            - "9121:9121"
```

```
    raftexample2:
        image: fsoppelsa/raftexample
        command: --id 2 --cluster
        http://127.0.0.1:9021,http://127.0.0.1:9022,
        http://127.0.0.1:9023 --port 9122
        ports:
            - "9022:9022"
            - "9122:9122"
    raftexample3:
        image: fsoppelsa/raftexample
        command: --id 3 --cluster
        http://127.0.0.1:9021,http://127.0.0.1:9022,
        http://127.0.0.1:9023 --port 9123
        ports:
            - "9023:9023"
            - "9123:9123"
```

该模板创建了三个 Raft 服务（raftexample1、raftexample2 和 raftexample3）。每个服务运行一个 raftexample 的实例，使用--port 暴露 API，并且使用--cluster 定义静态集群配置。

使用如下命令在 Docker 主机上启动集群：

```
docker-compose -f raftexample.yaml up
```

现在就可以开始实验了，比如杀死领导者，观察新的选举的进行，通过 API 设置一些值到其中一个容器里，移除容器，更新值，重启容器，获得值，验证该值已经正确升级了。

可以通过 curl 和 API 交互，详细信息位于：https://github.com/coreos/etcd/tree/master/contrib/raftexample。

```
curl -L http://127.0.0.1:9121/testkey -XPUT -d value
curl -L http://127.0.0.1:9121/testkey
```

本书不详细介绍这个实验，大家有兴趣可以自己试试。

尝试使用 Raft 实现时，为了保证性能最佳，可以选择 Etcd 的 Raft 库，为了开箱即用以及实现更简单的话，可以选择 Consul（来自 Serf 库）。

Etcd

Etcd 是一个高可用、分布式并且一致的键值存储，用来共享配置和服务发现。有一些著名项目使用 Etcd，包括 SwarmKit、Kubernetes 和 Fleet。

Etcd 能够在网络隔离的情况下优雅地管理 master 选举，并且能够容忍节点故障，包括 master 节点的故障。应用程序，本书指的是 Docker 容器和 Swarm 节点，能够读写 Etcd 的键值存储，比如服务位置等。

使用 Etcd 重新架构第 1 章示例

这里使用 Etcd 再次创建带有一个管理器和三个节点的示例。

本节使用一个真正的发现服务。通过在 Docker 里运行 Etcd 服务器来模拟一个非 HA 的系统。使用如下名称创建包含四个主机的集群：

- `etcd-m`：是 Swarm master，也用来托管 Etcd 服务器
- `etcd-1`：是第一个 Swarm 节点
- `etcd-2`：是第二个 Swarm 节点
- `etcd-3`：是第三个 Swarm 节点

运维人员可以连接到 `etcd-m: 3376`，从而在三个节点上操作 Swarm。

首先使用 Machine 创建主机：

```
for i in m `seq 1 3`; do docker-machine create -d virtualbox etcd-$i;
done
```

然后在 `etcd-m` 上运行 Etcd master。使用来自 CoreOS 的官方镜像 `quay.io/coreos/etcd`，具体步骤参见如下文档：https://github.com/coreos/etcd/blob/master/Documentation/op-guide/clustering.md。

首先在一个终端里，导入 etcd-m 的 shell 变量：

```
term0$ eval $(docker-machine env etcd-m)
```

然后，以单主机模式（这时，没有故障容忍等功能）运行 Etcd master：

```
docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
--name etcd quay.io/coreos/etcd \
-name etcd-m -initial-advertise-peer-urls http://$(docker-machine
ip etcd-m):2380 \
-listen-peer-urls http://0.0.0.0:2380 \
-listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
-advertise-client-urls http://$(docker-machine ip etcd-m):2379 \
-initial-cluster-token etcd-cluster-1 \
-initial-cluster etcd-m=http://$(docker-machine ip etcd-m):2380
-initial-cluster-state new
```

这里以 daemon（-d）模式启动 Etcd 镜像，并且暴露了端口 2379（Etcd 客户端通信端口）、2380（Etcd 服务器通信端口）和 4001()，同时指定如下 Etcd 参数：

- name：节点名称，本例中使用 etcd-m，作为托管该容器的节点名称
- initial-advertise-peer-urls，在本例的静态配置里，就是集群的地址：端口
- listen-peer-urls
- listen-client-urls
- advertise-client-urls
- initial-cluster-token
- initial-cluster
- initial-cluster-state

可以使用 etcdctl cluster-health 命令确认这个单节点的 Etcd 集群是健康的：

```
term0$ docker run fsoppelsa/etcdctl -C $(dm ip etcd-m):2379
cluster-health
```

```
$ docker run fsoppelsa/etcdctl \
> -C $(docker-machine ip etcdm):2379 \
> cluster-health

cluster is healthy
member e31763b19431cae7 is healthy
```

这表明 Etcd 至少已经启动并且正在运行了，可以使用它搭建 Swarm v1 集群。

在同一台 etcd-m 主机上创建 Swarm 管理器：

```
term0$ docker run -d -p 3376:3376 swarm manage \
-H tcp://0.0.0.0:3376 \`
etcd://$(docker-machine ip etcd-m)/swarm
```

这里会将主机的端口 3376 暴露给容器，但使用 etcd://URL 作为发现服务来启动管理器。

现在加入节点，etcd-1、 etcd-2 和 etcd-3。

和之前一样，每个终端导入一个节点的变量并且执行相应命令：

```
term1$ eval $(docker-machine env etcd-1)
term1$ docker run -d swarm join --advertise \
$(docker-machine ip etcd-1):2379 \
etcd://$(docker-machine ip etcd-m):2379
term2$ eval $(docker-machine env etcd-2)
term1$ docker run -d swarm join --advertise \
$(docker-machine ip etcd-2):2379 \
etcd://$(docker-machine ip etcd-m):2379
term3$ eval $(docker-machine env etcd-3)
term3$ docker run -d swarm join --advertise \
$(docker-machine ip etcd-3):2379 \
etcd://$(docker-machine ip etcd-m):2379
```

使用-advertise，让本地节点加入到 Swarm 集群里，使用在 etcd-m 上运行并且暴露服务的 Etcd 服务。

切换到 etcd-m 节点上，查看集群的节点情况，调用 Etcd 发现服务，如下所示：

```
$ docker run swarm list etcd://$(docker-machine ip etcdm):2379
time="2016-06-29T19:51:31Z" level=info msg="Initializing discovery without TLS"
192.168.99.106:2379
192.168.99.107:2379
192.168.99.109:2379
```

我们可以看到，已经有三台主机加入到集群里了。

ZooKeeper

ZooKeeper 是另一个分布式应用程序广泛使用且性能很好的协调性服务。Apache ZooKeeper 最初是 Hadoop 的子项目，但是现在已经是最高级的项目了。它是一个高度一致、可扩展并且可靠的键值存储，可以用作 Docker Swarm v1 集群的发现服务。如上文所述，ZooKeeper 使用 Paxos，而不是 Raft。

和 Etcd 类似，当 ZooKeeper 使用 quorum 组成节点集群时，它有一个领导者，其他节点则是跟随者。ZooKeeper 内部使用自己的 ZAB 和 ZooKeeper Broadcasting Protocol（ZooKeeper 广播协议），来保证一致性和完整性。

Consul

本章介绍的最后一种发现服务是 Consul，它是发现并且配置服务的工具。它提供 API，允许客户端注册并且发现服务。和 Etcd 以及 ZooKeeper 类似，Consul 也是键值存储，提供 REST API。它可以执行健康检查，决定服务的可用性，并且使用 Serf 库内的 Raft 共识算法。当然，和 Etcd 以及 ZooKeeper 类似，Consul 可以通过选举组成高可用的 quorum。它的成员管理系统基于 memberlist，这是一种高效的 Gossip 协议实现。

使用 Consul 重新架构第 1 章示例

本节将创建另一个 Swarm v1 集群，不过是在云供应商——DigitalOcean 上创建机器。要实现这一场景，需要一个访问令牌。但是，如果你没有 DigitalOcean 账号，可以使用 --driver virtualbox 代替--driver digitalocean，从而在本地运行本节示例。

先创建 Consul master：

```
$ docker-machine create --driver digitalocean consul-m
$ eval $(docker-machine env consul-m)
```

我们启动第一个代理。这里虽然称之为代理，但实际上是以 Server 模式运行的。在参数里指定使用 Server 模式（-server），并且设置其为 bootstrap 节点（-bootstrap）。使用这些参数的话，Consul 就不会执行选主程序，因为会强制该节点成为领导者。

```
$ docker run -d --name=consul --net=host \
consul agent \
-client=$(docker-machine ip consul-m) \
-bind=$(docker-machine ip consul-m) \
-server -bootstrap
```

如果为 HA 设定了 quorum，第二个和第三个节点必须使用参数-botstrap-expect 3 启动，让它们组成高可用集群。

使用 curl 命令验证 Consul quorum 启动成功了。

```
$ curl -X GET http://$(docker-machine ip consul-m):8500/v1/kv/
```

如果该命令没有显示任何错误的话，就意味着 Consul 工作正常。

接下来，在 DigitalOcean 上创建另外三个节点。

```
$ for i in `seq 1 3`; do docker-machine create -d digitalocean
consul-$i;
done
```

启动 master，并且使用 Consul 作为发现机制：

```
$ eval $(docker-machine env consul-m)
$ docker run -d -p 3376:3376 swarm manage \
-H tcp://0.0.0.0:3376 \
consul://$(docker-machine ip consul-m):8500/swarm
$ eval $(docker-machine env consul-1)
$ docker run -d swarm join \
  --advertise $(docker-machine ip consul-1):2376 \
consul://$(docker-machine ip consul-m):8500/swarm
$ eval $(docker-machine env consul-2)
$ docker run -d swarm join \
```

```
    --advertise $(docker-machine ip consul-2):2376 \
  consul://$(docker-machine ip consul-m):8500/swarm
  $ eval $(docker-machine env consul-3)
  $ docker run -d swarm join \
    --advertise $(docker-machine ip consul-3):2376 \
    consul://$(docker-machine ip consul-m):8500/swarm
```

如下是运行 `swarm list` 命令得到的结果，所有节点都已经加入了 Swarm，示例集群正在运行。

```
  $ docker run swarm list consul://$(docker-machine ip consulm):
8500/swarm
time="2016-07-01T21:45:18Z" level=info msg="Initializing discovery without
TLS"
  104.131.101.173:2376
  104.131.63.75:2376
  104.236.56.53:2376
```

实现去中心化的发现服务

Swarm v1 架构的限制在于它使用的是中心化的外部发现服务。这样的方案让每个代理都需要和外部的发现服务通信，发现服务所在的服务器的负载可能会呈指数级增长。在我们的实验里，对于一个 500 个节点的集群来说，建议使用至少由三台机器组成的 HA 发现服务，这些机器要有中高级别的配置、8 个核以及 8GB RAM。

为了合理解决这个局限问题，SwarmKit 和 Swarm Mode 所使用的发现服务设计为去中央化的，与 Swarm Mode 使用相同的发现服务代码库——Etcd，其位于所有节点上，没有单点故障。

本章小结

本章介绍了共识和发现服务的基本概念。了解了它们在编排集群里所起的至关重要的作用，它们提供容错和安全配置的服务。本章还详细分析了一种共识算法——Raft，并介绍了两种具体的 Raft 发现服务的实现：Etcd 和 Consul。并且实际演练了这两种服务的使用，使用它们重新架构了本书的基础示例。下一章我们会开始介绍使用内嵌 Etcd 库的 SwarmKit 和 Swarm。

第 3 章
遇见 Docker Swarm Mode

在 Dockercon 16 大会上，Docker 团队演示了一种运维 Swarm 集群的新方式，称为 Swarm Mode。大家对这一方式很是期待，因为推出了全新的工具集，也就是运维任意规模的分布式系统的工具，称为 **SwarmKit**。

本章要点：

- 介绍 SwarmKit
- 介绍 Swarm Mode
- 比较 Swarm v1、SwarmKit 和 Swarm Mode
- 创建一个测试用的 SwarmKit 集群，并在其上启动服务

阅读时请不要跳过 SwarmKit 一节，因为 SwarmKit 是 Swarm Mode 的基础。本书选择 SwarmKit 来介绍 Swarm Mode 的概念，比如节点、服务、任务等。

本书会在第 4 章“创建生产级别的 Swarm”里演示如何创建生产环境级别的大型 Swarm Mode 集群。

SwarmKit

Docker 团队在 DockerCon 16 大会上发布了 SwarmKit 和 Swarm Mode，SwarmKit 的定义为：

> “SwarmKit 是编排任意规模分布式系统的工具。它支持节点发现、基于 Raft 的共识、任务调度等。”

Swarm 集群由活动节点组成，这些节点可以作为管理器或者 worker 节点。

管理器，通过 Raft（如第 2 章“探索发现服务”所介绍的，在 quorum 可用时选择领导者）协调，负责分配资源、编排服务，以及在集群里分发任务。worker 节点则负责运行任务。

集群的目标是执行服务，因此在高层级上定义需要运行什么。比如，服务可能是“Web”服务。分配给节点的工作单元称为**任务**。分配给“Web”服务的任务可能是一个运行着 nginx 服务的容器，并且命名为 web.5。

一定要注意这里说的是服务，一个服务可能由多个容器一起提供。当然，这不是必须的。本书的重点当然是容器，但是 SwarmKit 的意图是理论上抽象任何对象的编排。

版本和支持

下文会介绍的 Docker Swarm Mode，其仅仅和 Docker 1.12+版本兼容。但是，使用 SwarmKit，可以编排之前版本的 Docker Engine，比如 1.11 版本和 1.10 版本。

SwarmKit 架构

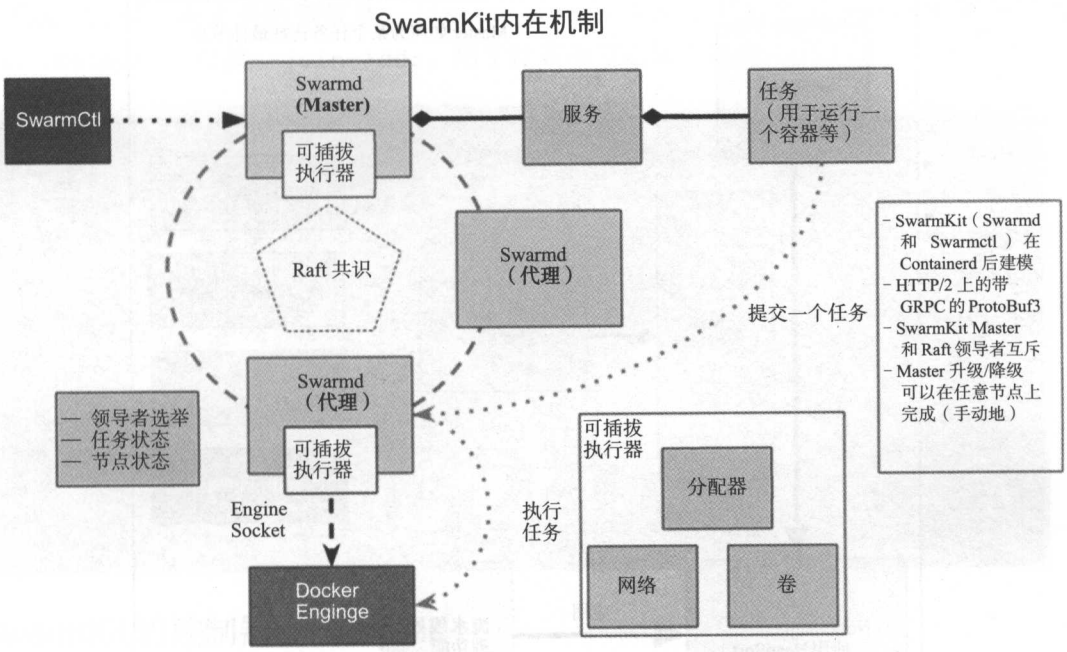
SwarmKit 是一种编排机制，用来处理任何规模的集群服务。

在 SwarmKit 集群里，节点要么是（集群）**管理器**，要么是 **worker 节点**（集群的工作节点，具体执行计算操作的节点）。

管理器的个数必须是奇数个，最好是 3 个或者 5 个，这样不会发生脑裂问题（第 2 章“探索发现服务”里详细介绍过），由多数管理器来驱动集群。Quorum 对于 Raft 共识算法来说是必须的。

SwarmKit 集群能够托管任意数量的 worker：1，10，100 或者 2000。

在管理器上，可以定义**服务**并且实现负载均衡。比如，服务可能是“Web”服务。“Web”服务物理上由几个**任务**组成，运行在集群节点，包括管理器上。其中的一个任务可能是单个 nginx Docker 容器。



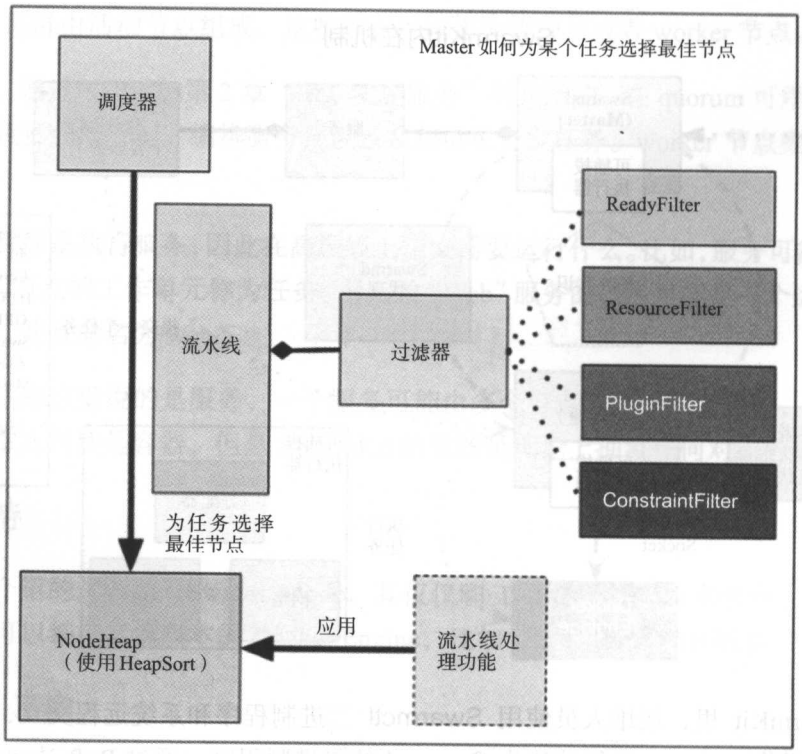
在 SwarmKit 里，运维人员使用 **Swarmctl** 二进制程序和系统远程交互，调用领导者 master 上的操作。Master 运行着称为 **Swarmd** 的二进制程序，通过 Raft 达成领导者共识、维护服务和任务的状态，并且调度 worker 节点执行 job。

worker 节点运行 Docker Engine，并且在单独的容器里运行 job。

SwarmKit 架构可以重新建模，但是核心组件（master 和 worker）会保留下来。可以通过插件方式加入新对象，用于分配资源，比如网络和卷。

管理器如何为某个任务选择最佳节点

SwarmKit 在集群里发起任务的方式称为**调度**。调度器是一种算法，使用过滤器之类的条件来决定物理上在哪里启动某个任务。



SwarmKit 的核心：swarmd

启动 SwarmKit 服务的核心二进制程序为 swarmd，这是创建 master 以及加入 slave 的后台程序。

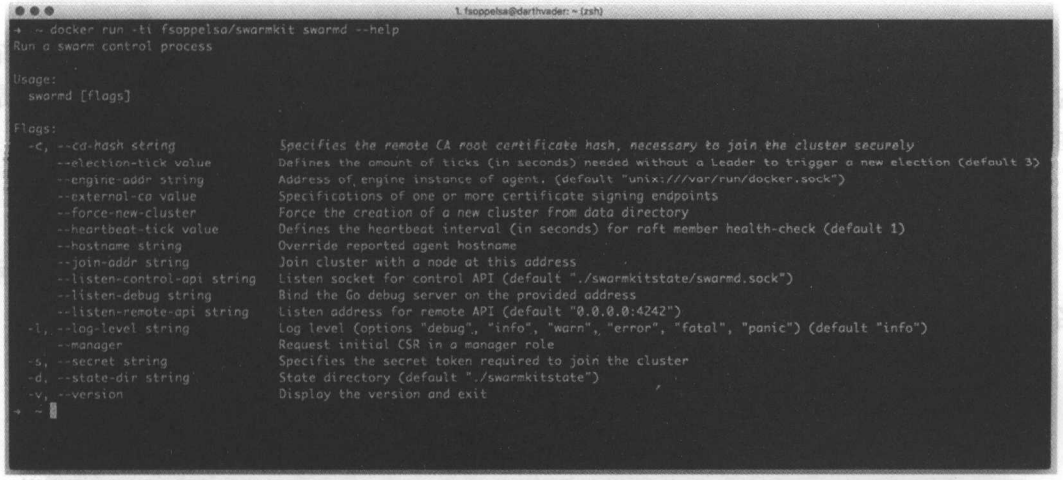
它可以将其自身绑定到本地 UNIX 套接字上，也可以绑定到 TCP 套接字上，不过在这两种情况下，都是由程序 `swarmctl` 管理的，实际连接到（另一个）专用 UNIX 本地套接字上。

下节示例使用 `swarmd` 创建第一个管理器，侦听端口 `4242/tcp`，然后在其他 worker 节点上再次使用 `swarmd`，将它们都加入管理器，最后使用 `swarmctl` 检查集群情况。

这些二进制程序封装在 `fsoppelsa/swarmkit` 镜像里，该镜像在 Docker Hub 上，这里使用它是为了简化并且避免 Go 代码的编译。

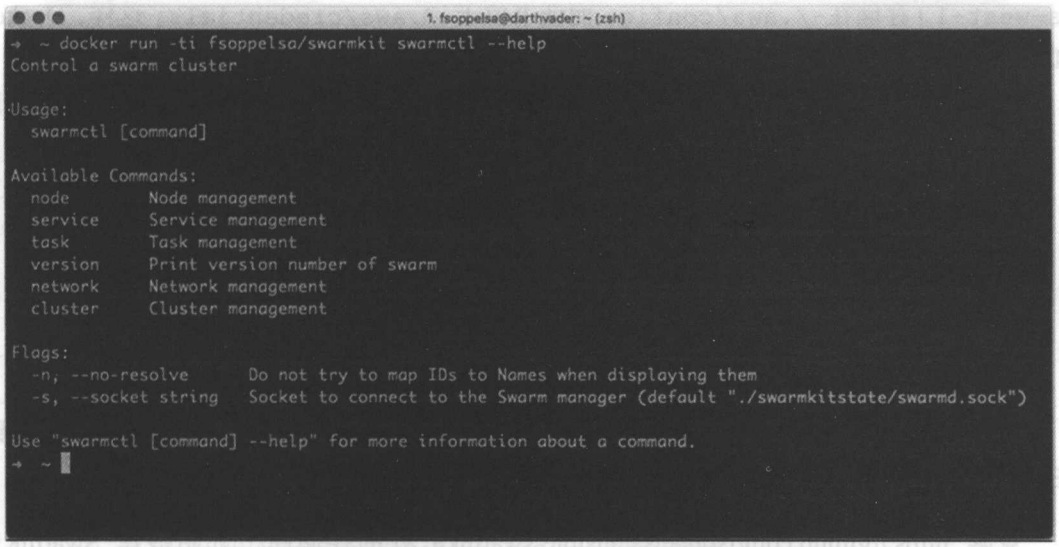
这是 swarmd 的帮助文档。参数的解释都很直观，因此本书不再详细介绍所有参数。

为了实验需要，最重要的参数是--listen-remote-api，其定义了 swarmd 要绑定的 address:port，以及--join-addr，用于让其他节点加入集群。



SwarmKit 的控制器：swarmctl

swarmctl 是 SwarmKit 的客户端部分。它是用来运维 SwarmKit 集群的工具，能够展示加入节点的列表、服务和任务列表，以及其他信息。还是在 fsoppelsa/swarmkit 里，swarmctl 的帮助文档为：



使用 Ansible 预配 SwarmKit 集群

本节会预配一个 SwarmKit 集群，由单个管理器和任意数量的 slave 组成。

要创建这样的集群，这里使用 Ansible 来稳定重复所需操作，并且根据 playbook 的结构介绍所用的命令。用户可以很轻松地在云供应商或者在本地使用这些 playbook，本节是在 Amazon 的 EC2 上做演示。

要运行本节示例，需要满足如下基本需求。

如果想要在 AWS 上执行本例，必须有一个 AWS 账号，并且配置了访问密钥。在**账号名/安全证书**处，通过 AWS Console 可以获得密钥。需要复制如下密钥的值：

- 访问密钥 ID
- 秘密访问密钥

可以使用 `awsctl` 设置这些密钥。如果使用的是 Mac 系统，则可以使用 *brew* 安装这个工具；如果使用的是 Linux 或 Windows 系统，可以使用自己的包系统来进行安装，之后完成配置：

```
aws configure
```

在需要时填入上述密钥。用户可以指定的配置信息包括，想要的 AWS region（例如 `us-west-1`）存储在`~/.aws/config`，而证书位于`~/.aws/credentials`。这样，Docker Machine 就配置好了，并且可以自动读取密钥。

如果想运行 Ansible 示例，而不去手动执行所有命令，需要如下软件：

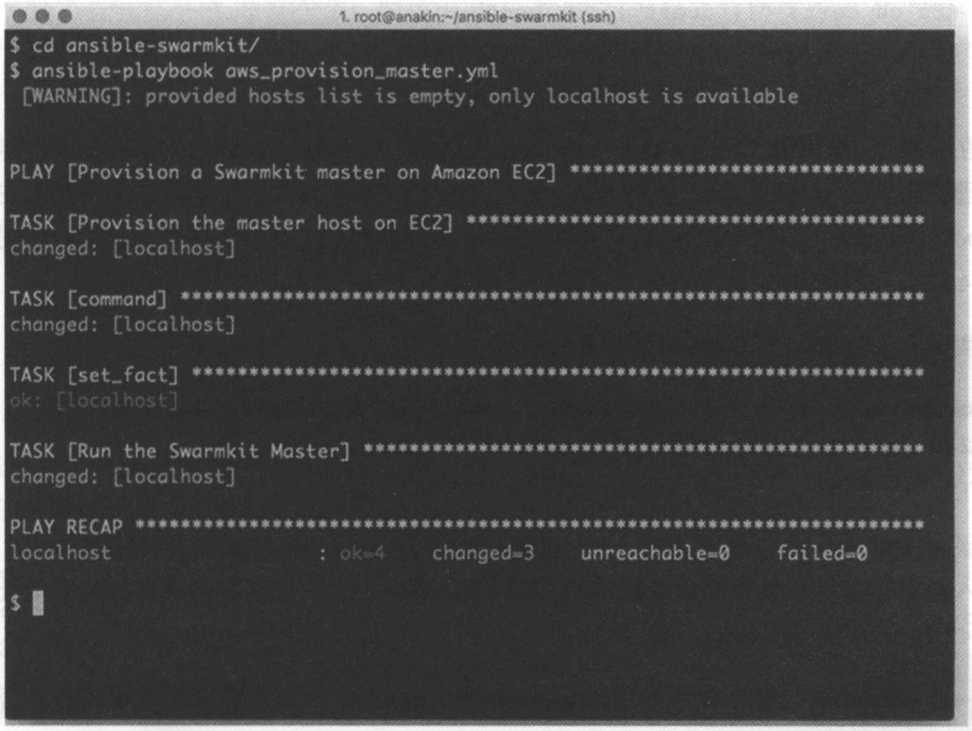
- Ansible 2.2+
- 和 Docker-machine 一起安装在 EC2 上的与镜像（本示例中默认为 Ubuntu 15.04 LTS）兼容的 Docker 客户端，撰写本书时是 Docker Client 1.11.2 版本
- Docker-machine
- Docker-py 客户端（Ansible 使用），可以通过 `pip install docker-py` 安装

另外，示例所使用的标准端口为 `4242/tcp`，用于集群节点之间的交互。因此，需要在安全组里开启该端口。

克隆 https://github.com/fsoppelsa/ansible-swarmkit 处的存储库，开始搭建 SwarmKit

Manager 节点：

```
ansible-playbook aws_provision_master.yml
```



Docker-machine 搭建之后，playbook 会在管理器主机上启动一个容器，作为 SwarmKit 管理器。这里是 playbook 的片段：

```
- name: Run the Swarmkit Master
  docker:
  name: swarmkit-master
  image: "fsoppelsa/swarmkit"
  command: swarmd --listen-remote-api 0.0.0.0:4242
  expose:
    - "4242"
  ports:
    - "0.0.0.0:4242:4242/tcp"
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
```

```
    detach: yes
    docker_url: "{{ dhost }}"
    use_tls: encrypt
    tls_ca_cert: "{{ dcert }}/ca.pem"
    tls_client_cert: "{{ dcert }}/cert.pem"
    tls_client_key: "{{ dcert }}/key.pem"
```

在主机上，从镜像 fsoppelsa/swarmkit 创建出的名为 swarmkit-master 的容器，以管理器模式运行 swarmd（侦听 0.0.0.0:4242）。swarmd 二进制程序直接使用主机上的 Docker Engine，因此 Engine 的套接字 mount 在容器内部。容器将端口 4242 映射到主机端口 4242，这样 slave 就可以通过连接到主机端口 4242 直接连接到 swarmd。

实际上，这和如下 Docker 命令是等价的：

```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock -p
4242:4242 fsoppelsa/swarmkit swarmd --listen-remote-api
0.0.0.0:4242
```

该命令以 detached 模式运行（-d），通过卷参数（-v）将 Docker 机器的 Docker 套接字传入到容器内部，从容器里暴露端口 4242 到主机（-p），让容器本身侦听任意地址，在端口 4242 上运行 swarmd。

一旦 playbook 完成，就可以导入 swarmkit-master 机器的证书，查看容器是否正确运行着：

```
1. root@anakin:~/ansible-swarmkit (ssh)
$ eval $(docker-machine env swarmkit-master)
$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED
    STATUS              PORTS                         NAMES
79d9be555dab        fsoppelsa/swarmkit     "swarmd --listen-remo"   3 minutes ago
    Up 36 seconds       0.0.0.0:4242->4242/tcp        swarmkit-master
$
```

现在是时候加入一些 slave 了。要启动一个 slave，仅仅需要运行：

```
ansible-playbook aws_provision_slave.yml
```

因为想要将多个节点加入到 SwarmKit 集群里，可以运行如下命令：

```
for i in $(seq 5); do ansible-playbook aws_provision_slave.yml;
done
```

该命令会运行五遍 playbook，创建出五个 worker 节点。在创建出名为 swarmkit-RANDOM 的机器后，playbook 会启动一个 fsoppelsa/swarmkit 容器，执行如下操作：

```
- name: Join the slave to the Swarmkit cluster
  docker:
    name: "{{machine_uuid}}"
    image: "fsoppelsa/swarmkit"
    command: swarmd --join-addr "{{ masterip }}":4242
    volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
    detach: yes
    docker_url: "{{ shost }}"
```

这里，swarmd 运行在 join 模式，并且通过连接端口 4242/tcp 加入到 Master 上初始化出的集群里。这和如下 docker 命令等价：

```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock
fsoppelsa/swarmkit swarmd --join-addr $(docker-machine ip swarmkitmaster):
4242
```

ansible 的 loop 命令需要一些时间才能完成，具体时间取决于需要启动多少 worker。当 playbook 完成时，就可以使用 swarmctl 控制这个集群了。如果你还没有导入 swarmkit-master 的机器证书，这里就该完成了：

```
eval $(docker-machine env swarmkit-master)
```

调用运行着 swarmd master 的容器，使用 exec 命令：

```
docker exec -ti 79d9be555dab swarmctl -s /swarmkitstate/swarmd.sock
node ls
```

```
1. root@anakin:~ (ssh)
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
        PORTS                    NAMES
79d9be555dab        fsoppelsa/swarmkit   "swarmd --listen-remo"   2 days ago          Up 2 days
        0.0.0.0:4242->4242/tcp   swarmkit-master
$ docker exec -ti 79d9be555dab swarmctl -s /swarmkitstate/swarmd.sock node ls
ID                         Name              Membership  Status  Availability  Manager Status
--                         ----              ----------  ------  ------------  --------------
3fm2amywezv6sy8p88blmssul  swarmkit-9f13f6d3  ACCEPTED    READY   ACTIVE
4qj8u9wtd2akkfoukv1bv66kq  swarmkit-ae4a3c86  ACCEPTED    READY   ACTIVE
549xqbs5trxtutk5204ifetj3  swarmkit-ba7d547e  ACCEPTED    READY   ACTIVE
7s1a8gcrm5wuerxl2egbp51ta  swarmkit-b4802b2f  ACCEPTED    READY   ACTIVE
bkab9maxaj1patti8nv0wibl3  swarmkit-1ad04abd  ACCEPTED    READY   ACTIVE
e9xscs6e7u4hrkdugepjaeroc  swarmkit-master    ACCEPTED    READY   ACTIVE       REACHABLE *
er1t2hm4vtq8pj1jhjv6vv71c  swarmkit-3ba2db01  ACCEPTED    READY   ACTIVE
$
```

上图可见，这里已经可以列出加入集群的 worker 列表。

在 SwarmKit 上创建服务

使用 `swarmctl` 二进制程序，可以创建一个服务（web），由 nginx 容器组成。

首先确保在这个全新的集群上没有活动的服务：

```
1. root@anakin:~ (ssh)
$ docker exec -ti 79d9be555dab swarmctl service ls
ID  Name  Image  Replicas
--  ----  -----  --------
$
```

使用如下命令启动一个服务：

```
docker exec -ti 79d9be555dab swarmctl service create --name web --
image nginx --replicas 5
```

```
1. root@anakin:~ (ssh)
$ docker exec -ti 79d9be555dab swarmctl service create --name web --image nginx --replicas 5
2ylsymmljv3t6bux43d7aqkba
$
```

该命令指定创建一个名为 `web` 的服务，由 `nginx` 容器镜像组成，并且使用因子 `5` 复制它，这样会在集群里创建 5 个 nginx 容器。需要几秒才能生效，因为在集群的每个节点上，Swarm 都会拉取并且启动 nginx 镜像，最终效果如下：

```
1. root@anakin:~ (ssh)
$ docker exec -ti 79d9be555dab swarmctl service ls
ID                         Name  Image  Replicas
--                         ----  -----  --------
2ylsymmljv3t6bux43d7aqkba  web   nginx  5/5
$
```

其中 5/5 意味着需要复制五个，已经启动了五个。使用 `swarmctl task ls` 可以看到这些容器的细节信息：

```
1. root@anakin:~ (ssh)
$ docker exec -ti 79d9be555dab swarmctl task ls
ID                         Service  Desired State  Last State               Node
--                         -------  -------------  ----------               ----
1a8itqd2nni4wwlig8pntfouw  web.5    RUNNING        RUNNING 5 minutes ago    swarmkit-master
5tt5yu3250ym492e481899l80  web.4    RUNNING        RUNNING 5 minutes ago    swarmkit-ae4a3c86
btf5lzk6w347wygbb82ch6su3  web.2    RUNNING        RUNNING 5 minutes ago    swarmkit-3ba2db01
cc267di5cz9ugjckn3ywacbvd  web.3    RUNNING        RUNNING 5 minutes ago    swarmkit-b4802b2f
e62tfxrb5b5fjy5axgxe2339l  web.1    RUNNING        RUNNING 5 minutes ago    swarmkit-9f13f6d3
$
```

不过等一下，nginx 服务(web.5)是运行在管理器节点上的么？是的。SwarmKit 和 Swarm Mode 管理器默认允许运行任务，调度器可以向其上分发 job。

在实际生产环境的配置里，如果想要预留管理器，不在其上运行 job，需要使用标签和约束条件的配置。在第 5 章“管理 Swarm 集群”里会详细介绍。

Swarm Mode

Docker Swarm Mode (Docker Engine 1.12 或更新版本) 导入了 SwarmKit 库，让跨多主机的分布式容器编排变成可能，并且易于操作。

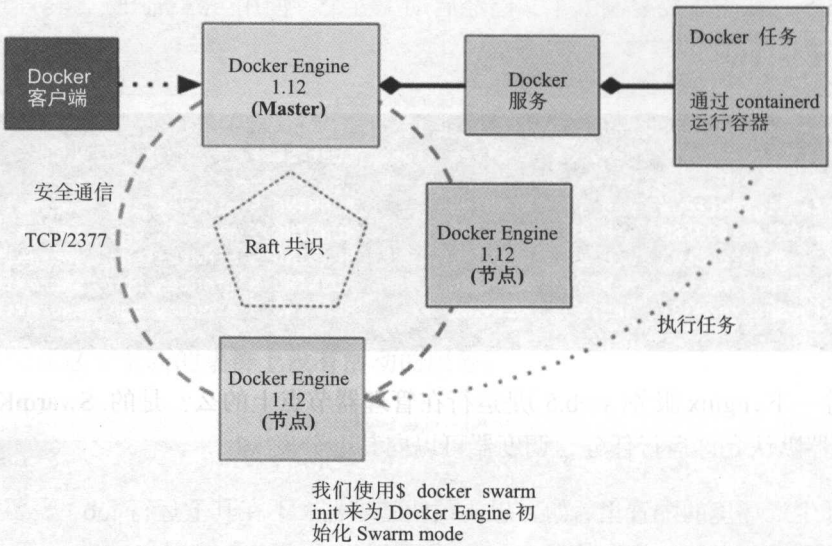
SwarmKit 和 Swarm Mode 的主要区别在于，从 1.12 版本开始，Swarm Mode 集成进了 Docker 本身。这意味着 Swarm Mode 命令，比如 `swarm`、`nodes`、`service` 和 `task` 在 Docker 客户端里就可以使用，并且通过 docker 命令可以初始化和管理 Swarm，可以部署服务和任务：

- `docker swarm init`：用来初始化 Swarm 集群

- `docker node ls`：用来列出可用节点
- `docker service tasks`：用来列出和某个特定服务相关的任务列表

Swarm v1 vs Swarm Mode vs SwarmKit

撰写本书时（2016 年 8 月），Docker 提供了三种 Docker 编排系统：Swarm v1、Swarm Mode 和 SwarmKit，也就是集成的 Swarm Mode。



最初的 Swarm v1，在第 1 章“欢迎来到 Docker Swarm”里已经介绍过，目前仍然可以使用，还没有弃用。它使用旧的基础架构。但是从 Docker 1.12 版本开始，Swarm Mode 是新的编排项目的推荐方案，特别适用于集群需要扩展到大规模的情况下。

这里使用表格简明扼要地总结了这些项目之间的区别。

首先是，Swarm v1 vs Swarm Mode：

Swarm standalone	Swarm Mode
Docker 1.8 版本后可用	Docker 1.12 版本后可用
作为容器可用	集成进 Docker Engine 里
需要外部发现服务（比如 Consul、Etcd、或者 ZooKeeper）	不需要外部发现服务，集成了 Etcd

续表

Swarm standalone	Swarm Mode
默认是不安全的	默认是安全的
副本和扩展特性不可用	副本和扩展特性可用
没有建模微服务的服务和任务的概念	有开箱即用的服务、任务、负载均衡和服务发现
没有额外可用的网络	集成了 VxLAN（mesh 网络）

再比较 SwarmKit 和 Swarm Mode：

SwarmKit	Swarm Mode
作为二进制程序（`swarmd` 和 `swarmctl`）发布——使用 swarmctl	集成进了 Docker Engine 里——使用 docker
有通用任务	有容器任务
包括服务和任务	包括服务和任务
不包括任何服务高级特性，比如负载均衡和 VxLAN 网络	包括开箱即用的服务高级特性，比如负载均衡和 VxLAN 网络

深入了解 Swarm Mode 部署

上述表格已经总结了 Swarm standalone 和 Swarm Mode 的对比，Swarm Mode 里的主要新特性已经集成进了 engine，不需要外部的发现服务，并且已经包含了副本、扩展、负载均衡以及网络。

集成进 engine

在 Docker 1.12+版本里，Docker 客户端添加了一些新命令。这里介绍和本书主题相关的新命令。

docker swarm 命令

这是当前管理 Swarm 的命令：

```
➜ ~ docker swarm --help

Usage:  docker swarm COMMAND

Manage Docker Swarm

Options:
      --help   Print usage

Commands:
  init        Initialize a swarm
  join        Join a swarm as a node and/or manager
  join-token  Manage join tokens
  update      Update the swarm
  leave       Leave a swarm

Run 'docker swarm COMMAND --help' for more information on a command.
```

该命令接受如下参数：

- `init`：初始化 Swarm。该命令在后台的当前 Docker 主机上创建一个管理器，并且生成 *secret*（这是 worker 传递给 API 的密码，从而可以通过认证加入集群）。
- `join`：worker 使用该参数加入集群，必须指定 *secret*，以及管理器 IP 端口值列表。
- `join-token`：用来管理 `join-token`。`join-token` 是一种特别的令牌 secret，用于加入管理器或者 worker（管理器和 worker 有不同的令牌值）。该命令可以很方便地让 Swarm 打印出加入管理器或 worker 的必需命令：

```
docker swarm join-token worker
```

要将某个 worker 添加到 Swarm 里，可以运行如下命令：

```
docker swarm join \ --token SWMTKN-1-
36gj6g1gi3ub2i28ekm1b1er8aa51v1tv00760t7umh3wmo1sc-
aucj6a94tqhhn2k0iipnc6096
\ 192.168.65.2:2377
docker swarm join-token manager
```

要将某个管理器添加到 Swarm 里，可以运行如下命令：

```
docker swarm join \ --token SWMTKN-1-
36gj6glgi3ub2i28ekm1b1er8aa51vltv00760t7umh3wmo1sc-
98glton0ot50j1yn8eci48rvq \ 192.168.65.2:2377
```

- `update`：通过变更一些值来更新集群，比如，可以使用该参数指定证书端点的新 URL。
- `leave`：让当前节点离开集群。如果有什么东西阻止该操作的完成，可以加上 `--force` 参数。

docker 节点

这是处理 Swarm 节点的命令。用户必须从管理器里启动该命令，因此在使用前需要先连接到管理器里。

```
➜ ~ docker node --help

Usage:  docker node COMMAND

Manage Docker Swarm nodes

Options:
      --help   Print usage

Commands:
  demote      Demote a node from manager in the swarm
  inspect     Display detailed information on one or more nodes
  ls          List nodes in the swarm
  promote     Promote a node to a manager in the swarm
  rm          Remove a node from the swarm
  ps          List tasks running on a node
  update      Update a node

Run 'docker node COMMAND --help' for more information on a command.
```

- demote 和 promote：该命令用来管理节点状态。借助该机制，用户可以将某个节点升级为管理器，或者降级为 worker。实际上，Swarm 将尝试 demote/promote。本章后面会详细介绍这个概念。
- inspect：该参数和 docker info 作用一样，只不过是用来得到 Swarm 节点的信息。它会打印出节点相关的信息。
- ls：列出连接到该集群的节点。
- rm：尝试移除一个 worker。如果用户想要移除管理器，必须先将其降级为 worker。
- ps：显示运行在特定节点上的任务列表。
- update：让用户可以变更某个节点的一些配置值，也就是标签。

docker 服务

该命令管理运行在 Swarm 集群上的服务：

```
➜ ~ docker service --help

Usage:  docker service COMMAND

Manage Docker services

Options:
      --help   Print usage

Commands:
  create      Create a new service
  inspect     Display detailed information on one or more services
  ps          List the tasks of a service
  ls          List services
  rm          Remove a service
  scale       Scale one or multiple services
  update      Update a service

Run 'docker service COMMAND --help' for more information on a command.
```

可以看到，除了一些熟悉的命令，比如 create、inspect、ps、ls、rm 和 update，还有个有意思的命令：scale。

Docker Stack

stack 命令对于 Swarm 的运维来说并非必需，在 Docker 1.12 版本上作为实验性质引入。Stack 是一组容器。比如，nginx+php+mysql 容器可以放到一个自包含的 Docker Stack 里，称为**分布式应用程序组（DAB）**，通过 JSON 文件描述。

Docker Stack 的核心命令是部署，负责创建和更新 DAB。本书将在第 6 章“Swarm 上真实应用的部署”里详细介绍。

已经集成了 Etcd 的 Raft

Docker Swarm Mode 通过 CoreOS Etcd Raft 库集成了 Raft。不再需要集成外部的发现服务，比如 ZooKeeper 或者 Consul。Swarm 直接负责必需的服务，比如 DNS 和负载均衡。

安装 Swarm Mode 集群也就是启动 Docker 主机，运行 Docker 命令，非常简单。

负载均衡和 DNS

从设计上看，集群管理器使用 Docker 内部的 DNS，给 Swarm 里的每个服务指派唯一的 DNS 名称和负载均衡的运行容器。查询和解析自动开箱即用。

对于使用--name myservice 创建的服务来说，Swarm 里的每个容器都能够解析服务的 IP 地址，因为它们使用 Docker 内嵌的 DNS 服务器来解析（dig myservice）内部网络名称。因此，如果有一个 nginx-service（比如由 nginx 容器组成），用户就可以 ping nginx-service 来访问前端。

另外，在 Swarm Mode 里，运维人员负责将服务端口 publish 到外部的负载均衡器上。随后将端口暴露到外部，端口范围从 30000 到 32767。Swarm 内部使用 iptable 和 IPVS 来分别执行包过滤和转发，以及负载均衡。

Iptable 是 Linux 默认使用的包过滤防火墙，而 IPVS 是在 Linux 内核里定义的 IP 虚拟服务器，可以用来负载均衡流量，这也正是 Docker Swarm 使用它来做的事情。

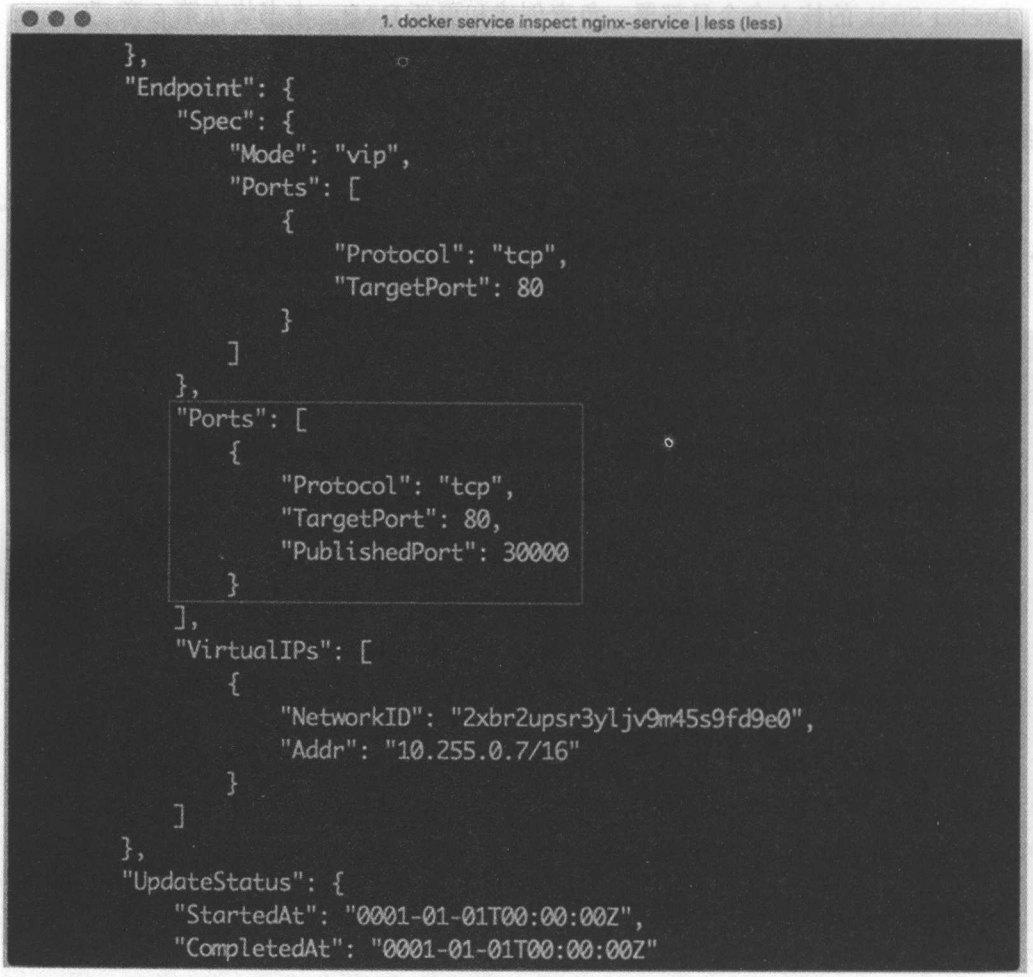
当新服务创建或者更新时可以使用--publish-add 参数发布端口。使用该参数，会发布一个负载均衡的内部服务。

比如，如果有一个包含三个 worker 的集群，每个 worker 都运行着 nginx（服务名为

nginx-service），就可以使用如下命令将它们的目标端口暴露给负载均衡器：

```
docker service update --port-add 80 nginx-service
```

这样会创建出任意节点集群的发布端口 30000 到 nginx 容器（端口 80）的映射。如果将任意节点连接到端口 30000，就会看到 Nginx 的欢迎页面。



这一切是怎么实现的呢？如上图所示，有一个关联的 VirtualIP（10.255.0.7/16），或者简称为 VIP，它位于 overlay 网络 **2xbr2upsr3yl** 上，这是由 Swarm 创建的，可以进入负载均衡器之中：

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b97618c9757a        bridge              bridge              local
6b28319e5581        docker_gwbridge     bridge              local
a11d270f8f75        host                host                local
2xbr2upsr3yl        ingress             overlay             swarm
43c936bc20d3        none                null                local
➜ ~
```

从任意主机都能够访问 `nginx-service`，因为 DNS 名称解析为 VIP，这里是 10.255.0.7，作为负载均衡器的前端。

在 Swarm 的每个节点上，Swarm 在内核里，特别是在命名空间里，实现了负载均衡，在该网络专用的网络命名空间里，Swarm 在 OUTPUT 链上添加了 MARK 规则，如下图所示：

```
Chain OUTPUT (policy ACCEPT 21 packets, 1634 bytes)
 pkts bytes target     prot opt in     out     source               destination
    4   336 MARK       all  --  *      *       0.0.0.0/0            10.255.0.7           MARK set 0x101
```

本书将在第 5 章“管理 Swarm 集群”和第 8 章“学习 Swarm 的额外特性”里详细介绍网络相关的概念。

升级和降级

使用 `docker node` 命令，集群运维人员可以将节点从 worker 升级到管理器，也可以反向将管理器降级为 worker。

将管理器降级为 worker 是将某个管理器（降级后就是 worker）从集群里移除的唯一方式。

本书将在第 5 章“管理 Swarm 集群”里详细介绍升级和降级的操作。

副本和扩展

在 Swarm 集群上部署应用程序，也就是定义以及配置服务、启动服务和等待 Docker engine 分发到集群里从而启动容器。本书会在第 6 章“Swarm 上真实应用的部署”里介绍如何在 Swarm 上部署完整的应用程序。

服务和任务

Swarm 工作负载的核心可以分解为各种服务。服务是一组任意数量的任务的抽象（这里的数量称为副本因子，或者副本数）。

任务是运行着的容器。

docker service scale

使用 `docker service scale` 命令，可以让 Swarm 确保集群上同时运行着特定数量的副本。比如，一开始集群上运行着相同任务的 10 个容器，然后需要将其规模扩展到 30 时，仅仅需要执行：

```
docker service scale myservice=30
```

Swam 需要调度 20 个新容器，它会基于负载均衡、DNS 以及网络一致性作出决策。如果任务的某个容器下线了，副本因子变为了 29，Swarm 就会在另一个集群节点上重新调度一个新容器（该容器会拥有一个新 ID），从而保证因子始终等于 30。

在添加副本和新节点中有一点需要注意。大家通常会问 Swarm 的自动能力是什么？如果有五个 worker 运行着 30 个任务，添加五个新节点，Swarm 并不会自动将这 30 个任务均衡到新的节点上，也就是说不会自动将它们从原来的节点挪到新节点上。Swarm 调度器的行为是保守的，直到某些事件（比如，运维人员干预）触发新的 `scale` 命令。仅仅在这种情况下，调度器才会考虑到这 5 个新节点，并且可能会在这 5 个新 worker 上启动新的副本任务。

本书在第 7 章“扩展平台”里会详细介绍 `scale` 命令的实际工作方式。

本章小结

本章介绍了 Docker 生态系统里的新成员：SwarmKit 和 Swarm Mode。从使用 Ansible 在 Amazon AWS 上创建 SwarmKit 集群的简单实现开始，随后介绍了 Swarm Mode 的核心概念，介绍了其接口和内部原理，包括 DNS、负载均衡、服务、副本以及升级/降级机制。现在，是时候深入了解真实的 Swarm Mode 部署了，在第 4 章“创建生产级别 Swarm”里将详细探讨。

第 4 章

创建生产级别 Swarm

本章将介绍如何创建包含上千个节点的真实 Swarm 集群，特别讨论如下话题：

- 部署大型 Swarm 的工具
- Swarm2k：构建出的最大型 Swarm Mode 集群之一，有 2300 个节点
- Swarm3k：第二个实验，有 4700 个节点的集群
- 如何规划硬件资源
- HA 集群拓扑
- Swarm 基础架构管理、网络以及安全
- 监控仪表盘
- 从 Swarm2k 和 Swarm3k 实验里学习到的经验教训

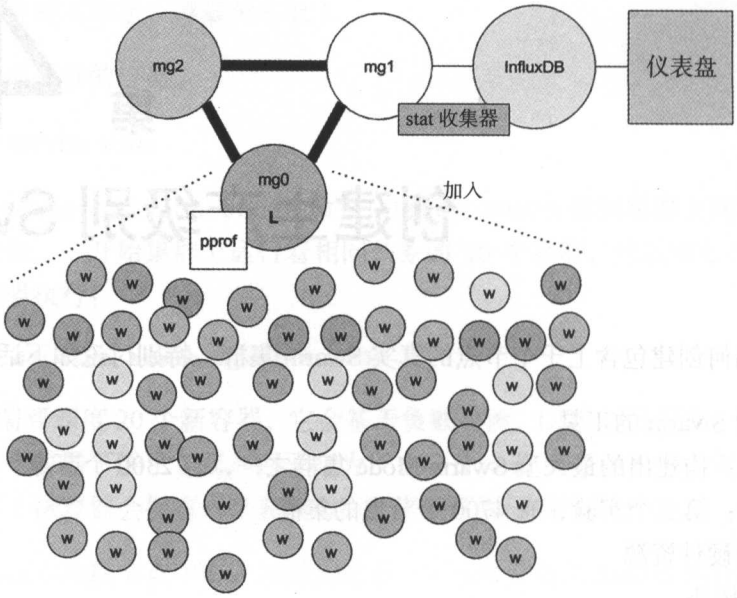
工具

使用 Swarm Mode，可以轻松地设计出生产级别的集群。

这里介绍的原则和架构很重要，揭示了不管使用的是什么工具，应该如何架构生产环境的安装。但是，从实际操作的角度来看，使用什么工具也很重要。

撰写本书时，Docker Machine 并非搭建大型 Swarm 的理想工具，因此本书介绍的生产级别部署时使用的工具是随本书开发的一个工具，已经在第 1 章“欢迎来到 Docker Swarm”里介绍过：belt（https://github.com/chanwit/belt）。本书将它和 Docker Machine、Docker Networking 以及 DigitalOcean 的 `doctl` 命令联合起来使用。

在第 5 章“管理 Swarm 集群”里我们将学习如何让 Swarm 的自动化创建成为可能。特别是，如何使用脚本以及其他机制，比如 Ansible，快速加入大量 worker。



Swarm2k 的 HA 拓扑

Swarm2k 和 Swarm3k 是合作式的实验。我们众筹了 Docker Host，而不是内存，号召大家参与。结果很令人吃惊——很多个人和公司，以及地理位置上很分散的贡献者加入了 Swarm2k 和 Swarm3k 项目。Swarm2k 总计收集到了大概 2300 个节点，而 Swarm3k 收集了 4700 个节点。

这里讨论 *Swarm2k* 的架构。上图中，有三个管理器，标示为 mg0、mg1 和 mg2。这里使用三个管理器是因为这是 Docker 核心团队推荐的管理器的最优数量。管理器在高速网络链接上组成 quorum，Raft 节点需要大量资源来同步其活动。因此，这里将管理器部署在同一个数据中心的 40GB 以太网链接上。

实验开始时，配置如下：

- mg0 是集群的管理器领导者

- mg1 托管 stat 收集器
- mg2 是 ready（预留）管理器

而 W 节点表示 Swarm worker。

安装在 mg1 上的 stat 收集器查询本地 Docker Engine 之外的信息，并且将其发送和存储到远程的时间序列的数据库——*InfluxDB* 里。选择 InfluxDB 是因为所用的监控代理——*Telegraf*，原生支持它。要展示集群的统计数据，这里使用 *Grafana* 作为仪表盘，下文会详细介绍。

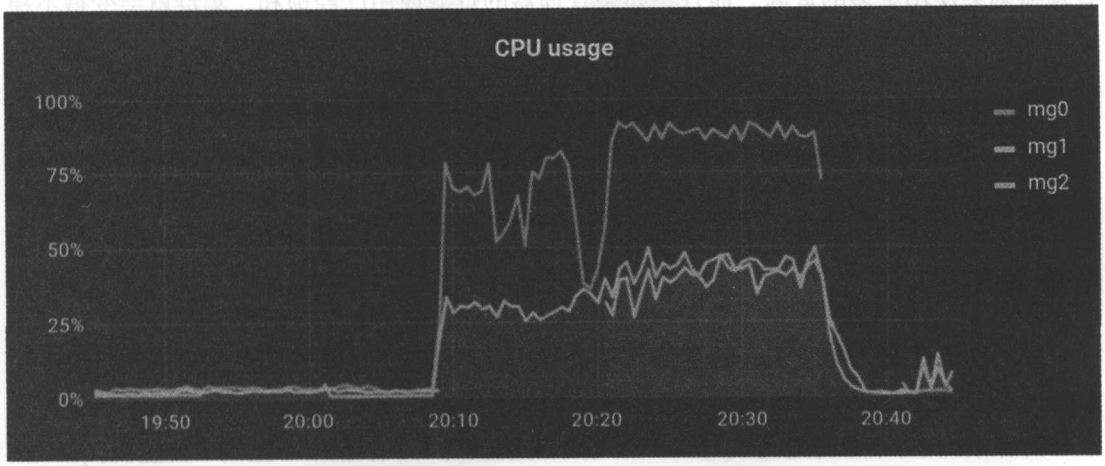
管理器配置

管理器是 CPU 密集型的，而不是内存密集型的。对于一个 500~1000 个节点的 Swarm 集群来说，我们观察到实际上 3 个 8 vCPU 的管理器就足够支撑相应的负载。但是，如果超过 2000 个节点，则推荐每个管理器至少有 16~20 个 vCPU 才能支撑起可能的 Raft 恢复。

Raft 恢复场景

下图展示了硬件升级中，进行大量 worker join 操作时的 CPU 使用情况。硬件升级到 8 vCPU（机器下线时间表示为连线的中断），当 mg1 和 mg2 重新加入集群时，可以看到 mg0，领导者的 CPU 使用跳涨到 75%~95%。导致这样跳涨的原因是 Raft 日志的同步和恢复。

在常规情况下，没有恢复发生时，每个管理器的 CPU 持续使用率都较低，如下图所示。



Raft 文件

在管理器主机上，Swarm 数据保存在 `/var/lib/docker/swarm` 里，称为 *swarm* 目录。Raft 数据保存在 `/var/lib/docker/swarm/raft` 里，并且包含 Write Ahead Log（WAL）和快照文件。

在这些文件里，有节点、服务和任务的实体，由 Protobuf 格式定义。

WAL 和快照文件会频繁地写入磁盘。在 SwarmKit 和 Docker Swarm Mode 里，每 10000 个实体会写入磁盘一次。因为这样的行为特点，我们将 swarm 目录映射到吞吐量大的快速专用磁盘上，通常是 SSD 磁盘。

本书会在第 5 章“管理 Swarm 集群”里详细介绍 swarm 目录损坏的情况下备份和恢复的流程。

运行任务

Swarm 集群的目标是运行服务，比如，由海量容器组成的大规模 Web 应用程序。本书将这种部署类型称为 *Mono* 模型。该模型里，网络端口是必须全局发布的资源。在 Docker Swarm Mode 的未来版本里会借助命名空间，让部署可以使用 *Multi* 模型，允许拥有多个子集群，为不同服务暴露相同的端口。

在小规模集群里，可以谨慎地让管理器同时负责托管 worker 任务。而在大型集群里，管理器则会使用更多的资源。而且，如果管理器的资源使用已经饱和，集群就会变得不稳定，变得反应迟钝，而不再接收任何命令。我们称这种状态为 *Berserkstate*。

要构建稳定的大型集群，比如 Swarm2k 和 Swarm3k，所有管理器必须设置为“Drained（疏散）”状态，这样任何任务都不会调度到管理器上，而只会在 worker 上调度：

```
docker node update --availability drain node-name
```

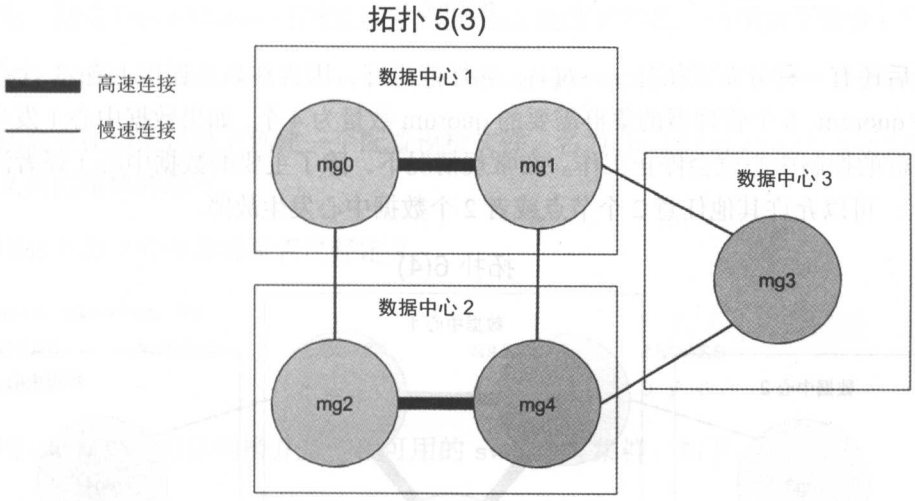
管理器拓扑

本书在第 5 章“管理 Swarm 集群”里会探讨 HA 属性，这里先简单介绍以便引入一些 Swarm 拓扑理论。HA 理论要求必须使用奇数个节点组成 HA 集群。下表展示了单个数据中心的容错因子。本章称之为 5(1)-3-2 公式，集群规模为 1 个数据中心里的 5 台主机，quorum

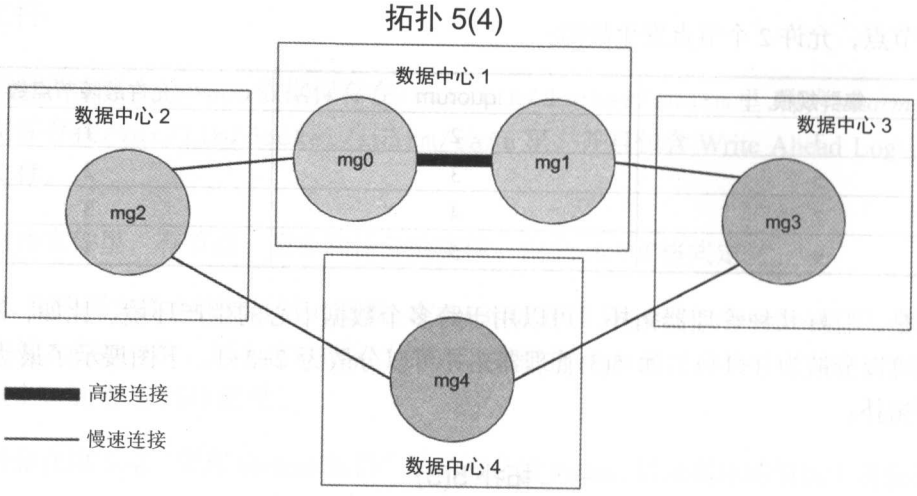
为 3 个节点，允许 2 个节点发生故障：

集群规模	quorum	允许故障节点数
3	2	1
5	3	2
7	4	3
9	5	4

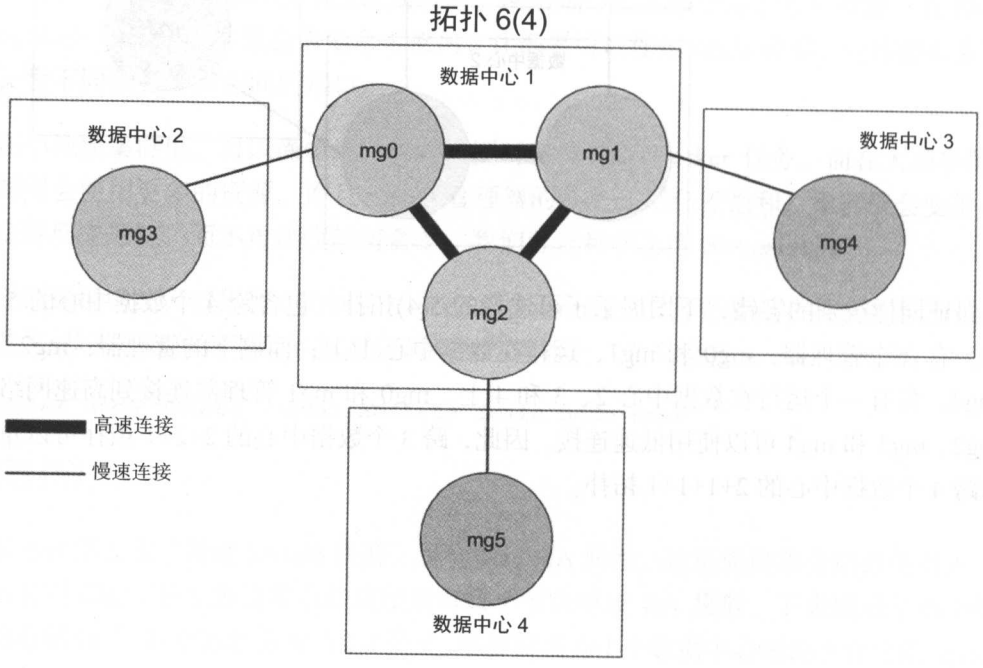
但是，有好几种管理器拓扑，可以用于跨多个数据中心的生产环境。比如，3(3)管理器拓扑可以分散为 1+1+1，而 5(3)管理器拓扑可以分散为 2+2+1。下图展示了最优的 5(3)管理器拓扑：



保证同样级别的容错，下图展示了可选择的 5(4)拓扑，包含跨 4 个数据中心的 5 个管理器。有 2 个管理器：mg0 和 mg1，运行在数据中心 1 上；而剩下的管理器：mg2、mg3 和 mg4，各有一个运行在数据中心 2、3 和 4 上。mg0 和 mg1 管理器连接到高速网络上，而 mg2、mg3 和 mg4 可以使用低速连接。因此，跨 3 个数据中心的 2+2+1 拓扑可以重新规划为跨 4 个数据中心的 2+1+1+1 拓扑。



最后还有一种分布式拓扑——6(4)，它性能更好，因为在高速连接上有 3 个节点组成了中央 quorum。6 个管理器的集群需要的 quorum 数量为 4 个。如果数据中心 1 发生故障，那么集群的控制中心就会停止工作。在常规情况下，除了主要的数据中心（译者注：数据中心 1），可以允许其他任意 2 个节点或者 2 个数据中心发生故障。



总的来说，尽可能保证存在奇数个管理器。如果强调管理器 quorum 的稳定性，就在高速连接上组成 quorum。如果想要避免单点故障，就尽可能将其分散。

要确定哪种拓扑是最合适的，可以尝试组成这些拓扑，故意让一些管理器下线来测试管理器的延迟，随后评估恢复的速度。

对于 Swarm2k 和 Swarm3k 来说，我们选择所有三个管理器都在同一个数据中心上的拓扑，因为这里想要获得最佳的性能。

使用 belt 预配基础架构

首先，创建 DigitalOcean 可用的名为 `swarm2k` 的集群模板，使用如下命令：

```
$ belt cluster new --driver digitalocean swarm2k
```

上述命令在当前目录下创建一个配置模板文件，名为`.belt/swarm2k/config.yml`。这是定义其他属性的起点。

使用如下命令检查集群是否已经定义：

```
$ belt cluster ls
CLUSTER      ACTIVE      LEADER      MASTERS      #NODES
swarm2k      -           -           -            0 / 0
```

使用 use 命令，可以切换并且使用可用的 `swarm2k` 集群，如下：

```
$ belt use swarm2k
swarm2k
```

这里定义 `swarm2k` 模板属性。

通过如下命令设置 DigitalOcean 实例 region 为 `sgp1`：

```
$ belt cluster update region=sgp1
```

belt 要求使用上述命令定义所有必需的值。如下是 `config.yml` 文件里指定的 DigitalOcean 驱动所需的模板关键字列表：

- `image`：指定 DigitalOcean 镜像 ID 或者快照 ID

- region：指定 DigitalOcean 的 region，比如 sgp1 或 nyc3
- ssh_key_fingerprint：指定 DigitalOcean 的 SSH 密钥 ID 或者 fingerprint
- ssh_user：指定镜像使用的用户名，比如 root
- access_token：指定 DigitalOcean 的访问令牌；推荐不把任何令牌放置在这里

每个模板属性都有其对应的环境变量。比如，access_token 属性可以通过 DIGITALOCEAN_ACCESS_TOKEN 设置。因此，实际场景里，还可以在开始操作前导出 DIGITALOCEAN_ACCESS_ TOKEN 作为 shell 变量。

配置完成后，运行如下代码验证当前模板属性：

```
$ belt cluster config
digitalocean:
  image: "123456"
  region: sgp1
  ssh_key_fingerprint: "800000"
  ssh_user: root
```

现在，使用如下命令创建 3 个 512MB 的管理器节点，称为 mg0、mg1 和 mg2：

```
$ belt  create 8192MB mg[0:2]
NAME    IPv4         MEMORY  REGION  IMAGE   STATUS
mg2     128.*.*.11   8192    sgp1    Ubuntu  docker-1.12.1 new
mg1     128.*.*.220  8192    sgp1    Ubuntu  docker-1.12.1 new
mg0     128.*.*.21   8192    sgp1    Ubuntu  docker-1.12.1 new
```

所有新节点都已经初始化了，状态为 new（新建）。

使用如下命令，等待所有 3 个节点成为活动节点：

```
$ belt  status  --wait  active=3
STATUS   #NODES   NAMES
new          3    mg2, mg1, mg0
STATUS   #NODES   NAMES
new          3    mg2, mg1, mg0
STATUS   #NODES   NAMES
```

```
new              3        mg2, mg1, mg0
STATUS     #NODES     NAMES
active           3        mg2, mg1, mg0
```

随后，将 node1 设置为活动的管理器主机，就可以组建 Swarm 了。使用如下命令将节点设置为活动节点：

```
$ belt active mg0
swarm2k/mg0
```

开始组建 Swarm。将 mg0 初始化为管理器领导者，如下：

```
$ belt docker swarm init --advertise-addr 128.*.*.220
Swarm initialized: current node (24j7sytbomhshtayt74lf7njo) is now
a manager.
```

上述命令输出的字符串可以复制粘贴，用来加入其他管理器和 worker，运行如下命令：

```
docker swarm join \
--token SWMTKN-1-1wwyxnfcgqt...fwzc1in3 \
128.*.*.220:2377
```

belt 提供了加入节点的快捷方式，语法如下，将 mg1 和 mg2 加入到 Swarm 里：

```
$ belt --host mg[1:2] docker swarm join \
--token --token SWMTKN-1-1wwyxnfcgqt...fwzc1in3 \
128.*.*.220:2377
```

至此，已经配置好了 mg0、mg1 和 mg2 管理器，可以加入 Swarm 的 worker 节点了。

使用 Docker Machine 保护管理器安全

Docker Machine 对于大量 Docker Engine 部署的扩展支持并不好，但是它很适合于自动保护小规模节点的安全。本节将使用 Docker Machine 来保护使用通用驱动的 Swarm 管理器的安全，该驱动允许用户控制已有主机。

本章示例已经在 mg0 搭建了 Docker Swarm 管理器。此外，我们通过启用远程端点的 TLS 连接来加强 Docker Engine 的安全。

Docker Machine 如何完成这一工作呢？首先，Docker Machine 通过 SSH 连接到主机上，检测到 mg0 的操作系统，示例为 Ubuntu；以及检测到预配器，示例为 systemd。

之后，会安装 Docker Engine。因为这里已经安装好了，所以会直接跳过这一步。

然后，也是最为重要的部分，它会生成一个 Root CA 证书，以及所有证书，并且将其存储到主机上。它还会自动配置 Docker 来使用这些证书。最终启动 Docker。

如果一切正常的话，Docker Engine 就会在启用 TLS 的情况下重新启动。

这里使用 Docker Machine 为 mg0、mg1 和 mg2 上的 Engine 生成 Root CA，并且配置 TLS 连接。然后使用 Docker 客户端进一步控制 Swarm，无须再使用更慢的 SSH。

```
$ docker-machine create \
  --driver generic \
  --generic-ip-address=$(belt ip mg0) mg0
Running pre-create checks...
Creating machine...
(mg0) No SSH key specified. Assuming an existing key at the default
location.
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Then we can test our working swarm with `docker info`. We grep only
15 lines for the brevity.
$ docker $(docker-machine config mg0) info | grep -A 15 Swarm
Swarm: active
 NodeID: 24j7sytbomhshtayt74lf7njo
 Is Manager: true
 ClusterID: 8rshkwfq4hsil2tdb3idpqdeg
 Managers: 3
```

```
 Nodes: 3
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
  CA Configuration:
  Expiry Duration: 3 months
```

另外，`docker node ls` 也可以正常工作。这里验证了 3 个管理器组成了最初的 Swarm，并且已经可以接受 worker 了：

```
$ docker $(docker-machine config mg0) node ls
ID                           HOSTNAME STATUS AVAILABILITY MANAGER STATUS
24j7sytbomhshtayt74lf7njo *  mg0      Ready  Active       Leader
2a4jcvp32aoa6olaxlelthkws    mg1      Ready  Active       Reachable
94po1ln0j0g5fgjnjfvm1w02r    mg2      Ready  Active       Reachable
```

集群的安全性如何？

我们使用 Docker 客户端连接启用 TLS 的 Docker Engine，并且在 Swarm 的节点之间也是 TLS 连接，CA 三个月后会过期，并且会自动循环。本书第 9 章“保护 Swarm 集群的安全以及 Docker 软件供应链里”会进一步讨论高级安全配置。

理解 Swarm 内部机制

这里创建了带有 3 个副本的 nginx 服务来验证 Swarm 是否有效：

```
$ eval $(docker-machine env mg0)
$ docker service create --name nginx --replicas 3 nginx
du2luca34cmy
```

在这之后，我们可以找到运行着的 Nginx 的 net 命名空间 ID。通过 SSH 连接到 mg0。

Swarm 的路由网格所用的网络命名空间和特别的网络命名空间拥有相同的时间戳，1-5t4znibozx。本示例中寻找的命名空间是 fe3714ca42d0。

```
root@mg0:~# ls /var/run/docker/netns -al
total 0
drwxr-xr-x 2 root root 120 Aug 22 15:38 .
drwx------ 5 root root 100 Aug 22 13:39 ..
-r--r--r-- 1 root root 0 Aug 22 15:17 1-5t4znibozx
-r--r--r-- 1 root root 0 Aug 22 15:36 d9ef48834a31
-r--r--r-- 1 root root 0 Aug 22 15:17 fe3714ca42d0
```

我们可以使用 ipvsadm 找到 IPVS 实体，并且使用 nsenter 工具（https://github.com/jpetazzo/ nsenter）在 net 命名空间里运行，如下：

```
root@node1:~# nsenter --net=/var/run/docker/netns/fe3714ca42d0 ipvsadm
-L
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
FWM 259 rr
  -> 10.255.0.8:0 Masq 1 0 2
```

这里注意存在一个活动的循环 IPVS 实体。IPVS 是内核级别的负载均衡器，Swarm 使用它和 iptable 的组合来平衡流量，iptable 用来转发以及过滤包。

在清除 nginx 测试服务（docker service rm nginx）之后，将管理器设置为 Drain（疏散）模式，从而避免它们执行任务：

```
$ docker node update --availability drain mg0
$ docker node update --availability drain mg1
$ docker node update --availability drain mg2
```

这里我们就可以在 Twitter 和 Github 上宣称管理器可以用了，就可以开始实验了！

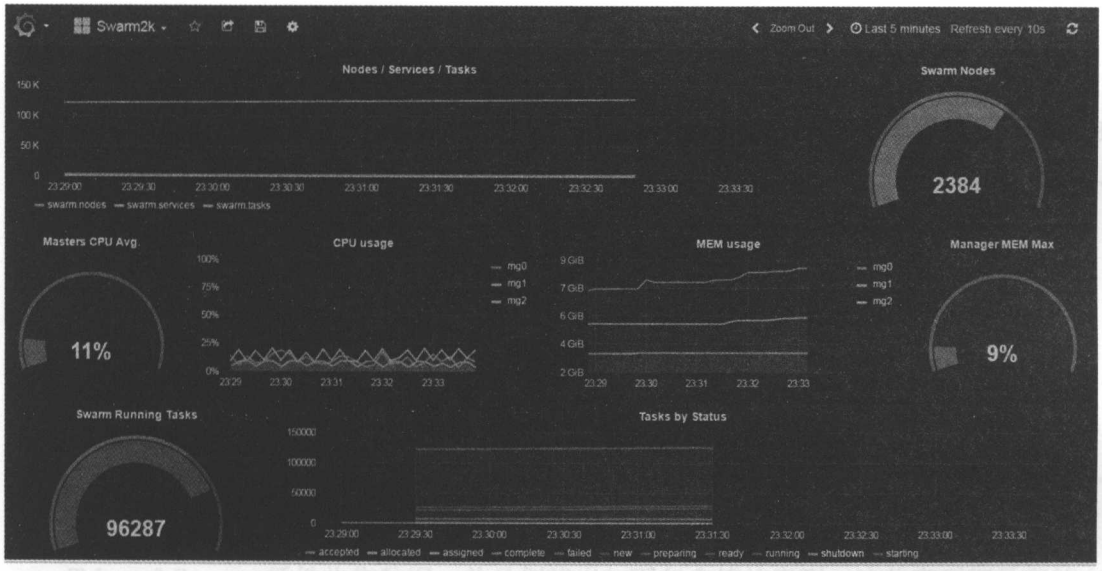
加入 worker

项目的贡献者们开始将其节点作为 worker 加入到管理器 mg0 上。大家可以使用各自喜欢的方法，具体为：

- 循环执行 `docker-machine ssh sudo docker swarm join` 命令
- Ansible
- 自定义的脚本和程序

本书会在第 5 章“管理 Swarm 集群”里详细介绍这些方法。

一段时间后，集群达到了 2300 个 worker 的 quota，并且启动一个 alpine 服务，副本因子为 100 000：



升级管理器

一段时间后，就会达到管理器的能力上限，我们不得不增加它们的物理资源。管理器的实时升级和维护是生产环境运维的必要能力。下面介绍如何完成这样的运维目标。

实时升级管理器

quorum 为奇数时，可以安全地降级某个管理器以作维护。

```
$ docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS
4viybni..h24zxde mg1 Ready Active Reachable
6xxwumb..j6zvtyg * mg0 Ready Active Leader
```

```
f1vs2e3..abdehnh mg2 Ready Active
```

mg1 是可以访问的管理器，这里使用如下命令将其降级为 worker：

```
$ docker node demote mg1
Manager mg1 demoted in the swarm.
```

我们可以看到当其变为 worker 时，mg1 的 Reachable 状态从 node1 的输出中消失了。

```
$ docker node ls
ID                  HOSTNAME    STATUS   AVAILABILITY   MANAGER      STATUS
4viybni..h24zxde    mg1         Ready    Active
6xxwumb..j6zvtyg *  mg0         Ready    Active         Leader
f1vs2e3..abdehnh    mg2         Ready    Active
```

当该节点不再是管理器时，就可以安全地关闭它，如下所示，使用 DigitalOcean 的 CLI：

```
$ doctl compute droplet-action shutdown 23362382
```

列出节点，我们可以看到 mg1 已经关闭了。

```
$ docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS
4viybni0ud2gjpay6ih24zxde mg1 Down Active
6xxwumbdac34bbgh6hj6zvtyg * mg0 Ready Active Leader
f1vs2e3hjiqjaukmjqabdehnh mg2 Ready Active
```

将其资源升级到 16GB 内存，随后再次启动该机器：

```
$ doctl -c .doctlcfg compute droplet-action power-on 23362382
```

这时在列出节点信息时，可能会有一些延时，因为 mg1 正在重新进入集群。

```
$ docker node ls
ID                  HOSTNAME    STATUS   AVAILABILITY   MANAGER  STATUS
4viybni..h24zxde    mg1         Ready    Active
6xxwumb..j6zvtyg *  mg0         Ready    Active         Leader
f1vs2e3..abdehnh    mg2         Ready    Active
```

最后，将其升级回管理器，如下所示：

```
$ docker node promote mg1
```

```
Node mg1 promoted to a manager in the swarm.
```

这时，集群恢复正常工作状态。可以对 mg0 和 mg2 重复上述操作。

监控 Swarm2k

生产级别的集群通常都需要一定程度的监控。目前，还没有特定的模式来监控 Swarm mode 下的 Docker 服务和任务。本书使用 Telegraf、InfluxDB 和 Grafana 来监控 Swarm2k。

InfluxDB 时间序列数据库

InfluxDB 是一种时间序列数据库，它没有任何依赖条件，因此很容易安装。InfluxDB 擅长存储度量、事件信息，以便之后分析使用。Swarm2k 使用 InfluxDB 存储集群、节点、事件以及使用 Telegraf 的任务的信息。

Telegraf 是可插拔的，有不少输入插件可以用来观测系统环境。

Telegraf Swarm 插件

本书开发了名为 Telegraf 的新插件，将 stat 存储到 InfluxDB 里。这个插件位于 http://github.com/chanwit/telegraf。数据包含值、标签和时间戳。基于时间戳计算或者聚合成值。另外，标签允许用户基于时间戳将这些值组合在一起。

Telegraf Swarm 插件收集数据，并且创建出如下包含值的序列，这些序列都是存储在 Swarm2k 里最有意义的标签以及 InfluxDB 里的时间戳：

- 序列 `swarm_node`：该序列包含 `cpu_shares` 和 `memory` 作为值，用户可以通过 `node_id` 和 `node_hostname` 标签来组合数据。
- 序列 `swarm`：该序列包含存储节点数量的 `n_nodes`、服务数量的 `n_services`，以及任务数量的 `n_tasks`。该序列不包含标签。
- 序列 `swarm_task_status`：该序列包含在某个时间点下通过状态组合的任务数量。该序列的标签为任务状态名称，比如，Started（已启动）、Running（正在运行）和 Failed（已失败）。

要启用 Telegraf Swarm 插件，需要在 `telegraf.conf` 里添加如下配置信息：

```
# Read metrics about swarm tasks and services
[[inputs.swarm]]
  # Docker Endpoint
  # To use TCP, set endpoint = "tcp://[ip]:[port]"
  # To use environment variables (ie, docker-machine), set endpoint =
    "ENV"
  endpoint = "unix:///var/run/docker.sock"
  timeout = “10s”
```

首先，搭建一个 InfluxDB 实例，如下：

```
$ docker run -d \
-p 8083:8083 \
-p 8086:8086 \
--expose 8090 \
--expose 8099 \
-e PRE_CREATE_DB=telegraf \
--name influxsrv
tutum/influxdb
```

然后，搭建 Grafana 实例，如下：

```
docker run  -d \
            -p 80:3000 \
            -e HTTP_USER=admin \
            -e HTTP_PASS=admin \
            -e INFLUXDB_HOST=$(belt ip influxdb) \
            -e INFLUXDB_PORT=8086 \
            -e INFLUXDB_NAME=telegraf \
            -e INFLUXDB_USER=root \
            -e INFLUXDB_PASS=root \
            --name grafana \
            grafana/grafana
```

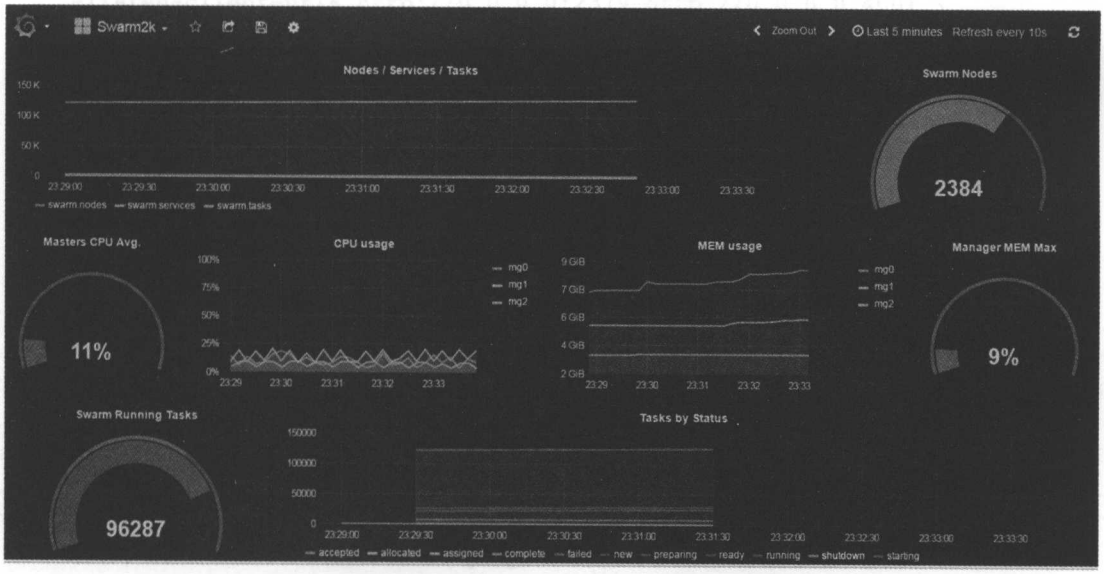
Grafana 实例搭建成功后，使用如下 JSON 配置文件创建仪表盘：

```
https://objects-us-west-1.dream.io/swarm2k/swarm2k_final_grafana_dashboard.
json
```

要将这个仪表盘连接到 InfluxDB 上，需要定义默认的数据源并且将其指向 InfluxDB 的主机端口 8086。如下是定义数据源的 JSON 配置文件。将$INFLUX_DB_IP 替换成你自己的 InfluxDB 实例。

```
{
        "name":"telegraf",
        "type":"influxdb",
        "access":"proxy",
        "url":"http://$INFLUX_DB_IP:8086",
        "user":"root",
        "password":"root",
        "database":"telegraf",
        "basicAuth":true,
        "basicAuthUser":"admin",
        "basicAuthPassword":"admin",
        "withCredentials":false,
        "isDefault":true
}
```

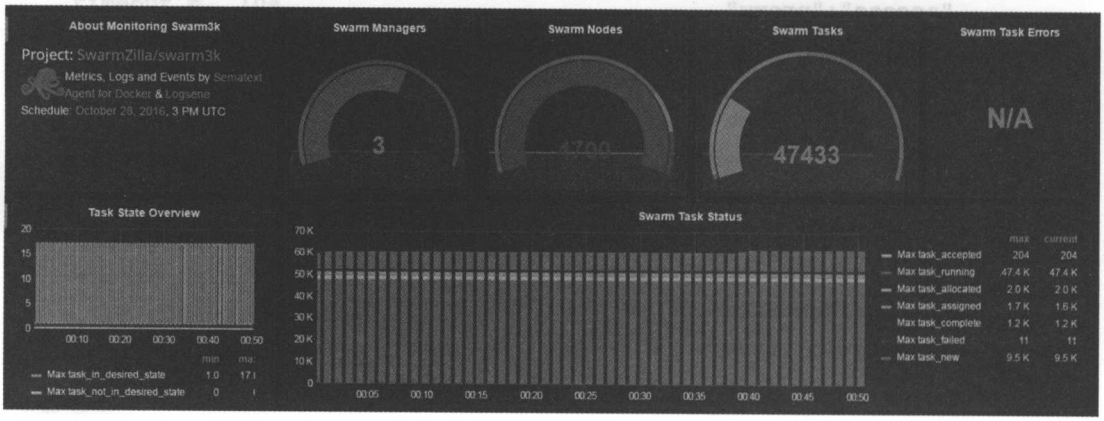
一切都连接好后，可以看到如下仪表盘：



Swarm3k

Swarm3k 是第二个合作式项目，其尝试使用 Swarm mode 构建更大的 Docker 集群。该项目开始于 2016 年 10 月 28 日，参加该项目的个人和企业超过 50 个。

Sematext 是第一个提供帮助的企业，其提供了 Docker 的监控和日志解决方案。这成为 Swarm3k 最终的官方监控系统。Stefan、Otis 及其团队从一开始就给我们提供了极大的支持。



Sematext 仪表盘

Sematext 是当时允许将监控代理部署为全局 Docker 服务的唯一一个 Docker 监控公司。这样的部署模型极大地简化了监控流程。

Swarm3k 的搭建和工作负载

Swarm3k 的目标是 3000 个节点，但是最终，我们成功构建了能够工作的、地理上分散的、拥有 4700 个节点的 Docker Swarm 集群。

管理器的配置为 128GB 内存的 DigitalOcean 节点，有 16 个 vCore，位于同一个数据中心里。

集群初始化配置包括没有记录的“KeepOldSnapshots”，这让 Swarm Mode 不用删除，而是保留所有数据快照以备后续分析。每个管理器的 Docker daemon 以 DEBUG 模式启动，从而能够得到更多运行中的信息。

我们使用 belt 搭建管理器，如前文所述，然后等待所有贡献者加入他们的 worker。

管理器使用的是 Docker 1.12.3 版本，而 worker 是 1.12.2 和 1.12.3 版本的混合。我们在 *ingress* 和 *overlay* 网络上组织服务。

我们一共计划了如下两种工作负载：

- 使用 MySQL 的 WordPress 集群
- C1M（Container-1-Million）

25 个节点组成一个 MySQL 集群。首先，创建 overlay 网络，`mydb`：

```
$ docker network create -d overlay mydb
```

然后，准备如下 `entrypoint.sh` 脚本：

```
#!/bin/bash
ETCD_SUBNET=${ETCD_SUBNET:-10.0.0.0}
ETCD_HOST=$(ip route get $ETCD_SUBNET | awk 'NR==1 {print $NF}')
/usr/local/bin/etcd \
 -name etcd0 \
 -advertise-client-urls
   http://${ETCD_HOST}:2379,http://${ETCD_HOST}:4001 \
 -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
 -initial-advertise-peer-urls http://${ETCD_HOST}:2380 \
 -listen-peer-urls http://0.0.0.0:2380 \
 -initial-cluster-token etcd-cluster-1 \
 -initial-cluster etcd0=http://${ETCD_HOST}:2380 \
 -initial-cluster-state new
```

然后，为特定版本的 Etcd 准备新的 Dockerfile，如下：

```
FROM quay.io/coreos/etcd
COPY entrypoint.sh /usr/local/bin/entrypoint.sh
RUN chmod +x /usr/local/bin/entrypoint.sh
ENTRYPOINT ['/usr/local/bin/entrypoint.sh']
```

开始使用之前，不要忘记使用`$ docker build -t chanwit/etcd`完成构建。

最后，将一个 Etcd 节点启动为 MySQL 集群的中央发现服务，如下：

```
$ docker service create --name etcd --network mydb chanwit/etcd
```

通过查看 Etcd 的 Virtual IP，可以得到服务的 VIP，如下：

```
$ docker service inspect etcd -f "{{ .Endpoint.VirtualIPs }}"
[{... 10.0.0.2/24}]
```

使用得到的信息，创建 mysql 服务，该服务可以扩展到任意规模。示例如下：

```
docker service create \
--name mysql \
-p 3306:3306 \
--network mydb \
--env MYSQL_ROOT_PASSWORD=mypassword \
--env DISCOVERY_SERVICE=10.0.0.2:2379 \
--env XTRABACKUP_PASSWORD=mypassword \
--env CLUSTER_NAME=galera \
--mount "type=bind,src=/var/lib/mysql,dst=/var/lib/mysql" \
perconalab/percona-xtradb-cluster:5.6
```

这里在 mynet 和 ingress 网络上都遇到了一些 IP 地址相关的问题，因为有一个 Libnetwork 的 bug，详情见 https://github.com/docker/docker/issues/24637。我们通过将集群仅仅绑定到单个 overlay 网络的 mydb 上来绕过了这个问题。

现在，尝试使用副本因子 1 来 docker service create WordPress 的容器。我们故意不去控制 WordPress 容器会调度到哪里。但是，当尝试将这个 WordPress 服务和 MySQL 服务连接时，连接持续超时。最后我们认为对于这种规模的 WordPress+MySQL 组合，最好在集群上添加一些限制条件，让所有服务都运行在相同的数据中心里。

大规模 Swarm 的性能

在这个问题里另外的收获是 overlay 网络的性能极大依赖于每台主机网络配置的正确调优。正如一名 Docker 工程师曾经建议的，当存在太多 ARP 请求时（网络很大时），我们可能会遇到“Neighbour Table Overflow”（邻居表溢出）错误，并且每台主机都无法正常回复。可以像如下这样调优每台 Docker 主机，从而修复这个问题：

```
net.ipv4.neigh.default.gc_thresh1 = 30000
net.ipv4.neigh.default.gc_thresh2 = 32000
net.ipv4.neigh.default.gc_thresh3 = 32768
```

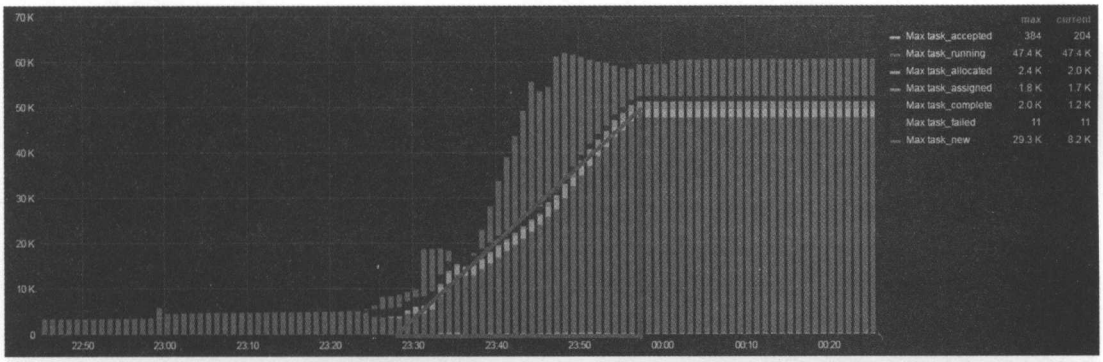
这里，gc_thresh1 是预计的主机数量，而 gc_thresh2 是软限制，gc_thresh3 是硬限制。

因此，当 MySQL+WordPress 测试失败时，我们改变了计划，在 Routing Mesh（路由网格）上实验 NGINX。

ingress 网络是使用/16 池来搭建的，因此它能够容纳最多 64000 个 IP 地址。基于 Alex Ellis 的建议，我们在集群上启动了 4000 个（四千个！）NGINX 容器。在测试过程中，持续有节点加入和退出。最终，几分钟后，NGINX 服务启动，并且路由网格构建成功。即使其中一些节点持续发生故障，它也能够正确提供服务，因此该测试验证了 1.12.3 版本的路由网格是非常健壮的，完全可以用于生产环境。我们随后停止了 NGINX 服务，并且开始测试尽可能多的容器的调度，目标数量为 1000000 个。

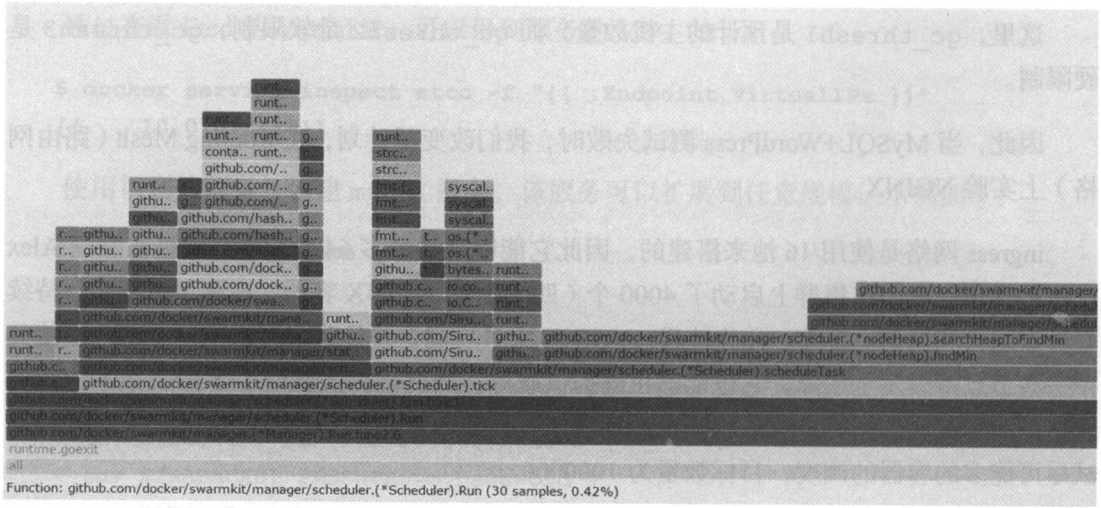
因此，和 Swarm2k 一样，我们创建了“alpine top”服务。但是这次调度时间有点慢。我们在大概 30 分钟达到了 47000 个容器。据此预测要在集群上启动 1000000 个容器大概需要 10.6 个小时。

因为预计要花这么长的时间，所以我们决定再次改变计划，改为启动 70000 个容器。

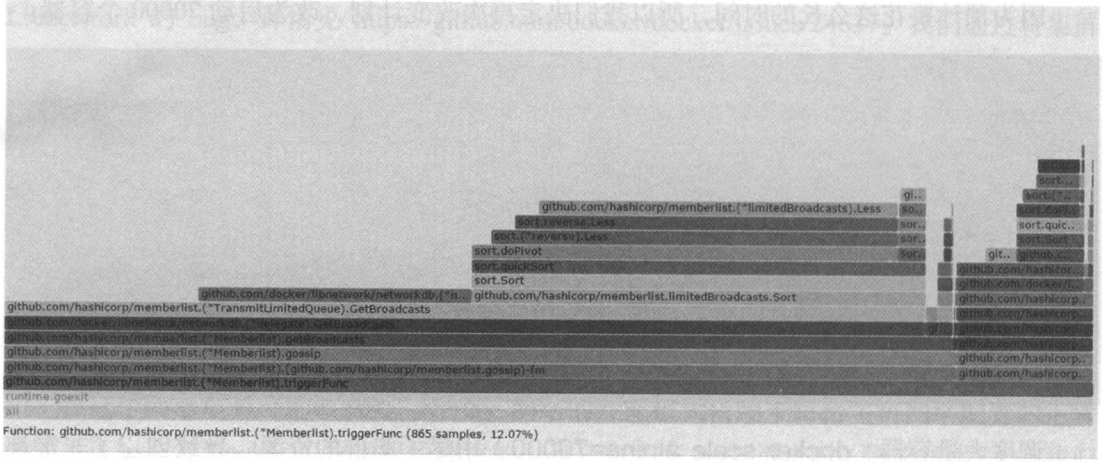


调度大量容器（docker scale alpine=70000）用尽了集群的资源。导致创建了非常长的尚未提交的调度队列，直到所有 70000 个容器都完成了调度。因此，当我们决定关闭管理器时，所有正在调度的任务就消失了，集群开始不稳定，Raft 日志崩溃了。

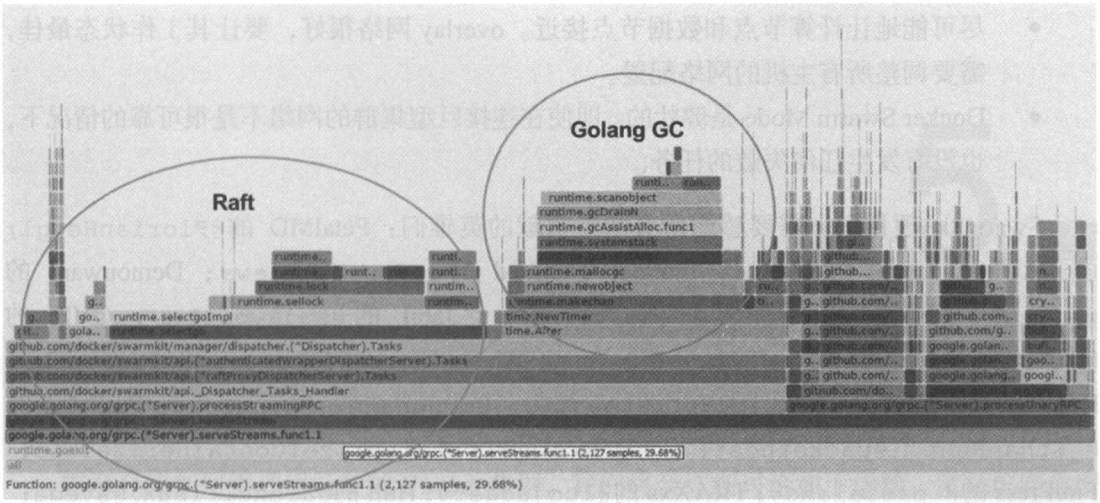
这一过程中最有意思的事情是，我们尝试收集 CPU profile 信息来查看哪个 Swarm primitive 给集群带来的压力最大。



这里我们可以看到仅仅 0.42%的 CPU 花在调度算法上。可以推论出 1.12 版本的 Docker Swarm 的调度算法极其快速。这意味着在 Swarm 的未来版本里，可以考虑在这里增加一些可接受的额外消耗，来引入更为精细的调度算法，从而带来更好的资源利用率。



另外，我们也发现大量 CPU 周期花在节点通信上。这里可以看到 Libnetwork 成员列表层。它占用了总体 CPU 的 12%。



不过，看上去最消耗 CPU 的是 Raft，我们也能看到其大量调用了 Go 垃圾回收器。其占用了总体 CPU 的 30%。

总结 Swarm2k 和 Swarm3k 的经验教训

如下是从这些实验里收获的经验教训：

- 在大量 worker 的情况下，管理器需要大量 CPU。无论何时发生 Raft 恢复程序，都会大量消耗 CPU。
- 如果领导管理器失效了，最好停止该节点上的 Docker，并且等待集群在 n-1 个管理器的情况下重新达到稳定状态。
- 让快照预留尽可能小。默认的 Docker Swarm 配置就可以实现这一点。持久化 Raft 的快照需要额外的 CPU。
- 上千个节点需要大量的资源，CPU 和网络带宽都需要很多。尽量保证服务和管理器拓扑物理上尽可能地紧凑。
- 成百上千的任务需要高内存的节点。
- 目前，稳定的生产环境推荐使用最多 500~1000 个节点。
- 如果管理器看上去卡住了，请等待；它们会最终恢复的。
- 要让路由网格工作，`advertise-addr` 参数是必需的。

- 尽可能地让计算节点和数据节点接近。overlay 网络很好，要让其工作状态最佳，需要调整所有主机的网络配置。
- Docker Swarm Mode 是健壮的。即使在连接巨型集群的网络不是很可靠的情况下，也没有发生任何失败的任务。

Swarm3k 项目上，需要感谢所有默默贡献的英雄们：PetalMD 的@FlorianHeigl; @jmaitrehenry；Rackspace，Internet Thailand 的@everett_toews；Demonware 的@squeaky_pl,@neverlock, @tomwillfixit；Jabil 的@sujaypillai；OVH 的@pilgrimstack;Collabnix 的@ajeetsraina;Aiyara Cluster 的@AorJoa 和@PNgoenthai; HotelQuickly 的@GroupSprint3r, @toughIQ, @mrnonaki, @zinuzoid; Packet.io 的@_EthanHunt_; @packethost; Conference 的@ContainerizeTContainerizeThis; FirePress 的@_pascalandy; TRAXxs 的@lucjuggery; Huli 的@alexellisuk; @svega; Emerging Technology Advisors 的@BretFisher; @voodootikigod；ThumpFlow 的@AlexPostID; @gianarb；Nipa Technology 的@Rucknar,@lherrerabenitez; @abhisak; NexwayGroup 的@djalal。

还要再次感谢 Sematext 提供了最佳的 Docker 监控系统，以及 DigitalOcean 提供了所有资源。

本章小结

本章介绍了在 DigitalOcean 上如何使用 belt 部署两个大型 Swarm 集群。这两个项目有很多可学习之处。本章总结了经验教训，并且提供了运行大型生产 Swarm 的建议和技巧。我们还介绍了一些 Swarm 特性，比如服务和安全，并且探讨了管理器拓扑。下一章会详细讨论如何管理 Swarm。介绍的话题包括使用 belt、脚本和 Ansible 部署 worker、管理节点、监控以及图形化界面。

第 5 章

管理 Swarm 集群

本章介绍如何管理运行着的 Swarm 集群。我们将会详细讨论如下话题，包括扩展集群规模（添加以及移除节点）、更新集群和节点信息、处理节点状态（升级和降级）、故障排查以及图形化界面（UI）。

本章将介绍如下主题：

- Docker Swarm standalone
- Docker Swarm Mode
- 集群管理
- Swarm 健康
- Swarm 的图形化界面

Docker Swarm standalone

在 standalone 模型下，集群运维需要在容器 Swarm 里直接完成。

这里不打算详细介绍所有参数。Swarm v1 很快就会被弃用，因为它已经被 Swarm Mode 取代了。

```
1. fsoppelsa@darthvader: ~ (zsh)
→  ~ docker run swarm --help
Usage: swarm [OPTIONS] COMMAND [arg...]

A Docker-native clustering system

Version: 1.2.4 (5d5f7f0)

Options:
  --debug                          debug mode [$DEBUG]
  --log-level, -l "info"           Log level (options: debug, info, warn, error, fatal, panic)
  --experimental                   enable experimental features
  --help, -h                       show help
  --version, -v                    print the version

Commands:
  create, c     Create a cluster
  list, l       List nodes in a cluster
  manage, m     Manage a docker cluster
  join, j       Join a docker cluster
  help          Shows a list of commands or help for one command

Run 'swarm COMMAND --help' for more information on a command.
→  ~
```

管理 Docker Swarm standalone 集群的命令如下：

- Create（c）：正如第 1 章“欢迎来到 Docker Swarm”里所介绍的，这是使用令牌机制时，如何生成 UUID 令牌的方式。
- List（l）：该命令会基于 Consul 或者 Etcd 做迭代，展示出集群节点列表。也就是说，必须将 Consul 或者 Etcd 作为参数传入。
- Join（j）：将运行着 Swarm 容器的节点加入集群。这里需要在命令行传入发现机制。
- Manage（m）：这是 standalone 模式的核心。管理集群涉及变更集群属性，比如过滤器、调度器、外部 CA 的 URL 以及超时时间。本书会在第 6 章“Swarm 上真实应用的部署”详细介绍这些参数。

Docker Swarm Mode

本节将继续探索 Swarm Mode 管理集群相关的命令。

手动添加节点

用户可以按照自己喜欢的方式，选择创建一个新的 Swarm 节点，或者 Docker 主机。

如果使用 Docker Machine，会很快达到其上限。用户在列举机器时必须非常有耐心，需要等待几秒钟 Machine 才能打印出所有信息。

手动添加节点的一种方式是使用带有通用驱动的 Machine，从而将主机预配（操作系统安装，网络和安全组配置等）委托给别的工具（比如 Ansible），并且之后使用 Machine 以合适的方式安装 Docker。过程如下：

1. 手动配置云环境（安全组，网络等）。
2. 使用第三方工具预配 Ubuntu 主机。
3. 在这些主机上运行带有通用驱动的 Machine，目的是正确安装 Docker。
4. 使用第 2 步的工具或者其他工具管理主机。

如果使用 Machine 的通用驱动，它会选择最新的稳定版 Docker 二进制文件。撰写本书时，为了使用 Docker 1.12 版本，有时候需要通过 Machine 获得最新的 Docker 的不稳定版本，我们使用的是`--engine-install-url`参数：

```
docker-machine create -d DRIVER --engine-install-url
https://test.docker.com mymachine
```

撰写本书时，对于生产环境的 Swarm（mode），1.12 版本是稳定的，因此不会经常需要使用上述命令，除非你需要使用一些最新的 Docker 特性。

管理器

当规划 Swarm 时，需要记住管理器数量相关的一些考虑因素，本书在第 4 章“创建生产级别 Swarm”介绍过。HA 理论建议管理器数量必须是奇数，等于或者大于 3。在高可用上设置 quorum 值意味着由多数节点的统一意见来决策运维操作。

如果有两个管理器，其中一个下线后又恢复了，可能这两个管理器都会被认为是领导者。这会导致集群组织架构的逻辑崩溃，称为结果分裂。

集群拥有的管理器越多，就越能减少故障的发生，如下表所示。

管理器数量	Quorum（多数）	最大可能故障数
3	2	1
5	3	2
7	4	3
9	5	4

另外，在 Swarm Mode 里，会自动创建 ingress overlay 网络，并且将其关联到节点以供 ingress 流量所用。其目的是供容器使用：

```
2. fsoppelsa@yoda: ~ (zsh)

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-42mf2wtk9m5pctdqiy5bcfqorj9zlnh5n1z3sszoia7l15jwtd-bo1vg5gilk7
lvjwibtx4x0ik0 \
    192.168.99.100:2377
➜ ~ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
223104f873c6        bridge              bridge              local
8b2dcbc553f3        docker_gwbridge     bridge              local
f3ee62fb8d94        host                host                local
2qf4nt5at8xl        ingress             overlay             swarm
91f8b56fd179        none                null                local
➜ ~
```

用户需要将容器关联到内部 overlay（VxLAN meshed）网络以便互相通信，而不是使用公开或者其他外部网络。因此，Swarm 为用户创建了这样的网络。

Worker 数量

用户可以添加任意数量的 worker。这是 Swarm 弹性的体现。完全可以有 5、15、200、2300 个或者 4700 个运行着的 worker。这是最容易处理的部分，用户可以随时随地，以任意规模，添加或者移出 worker。

添加脚本化节点

如果并非计划达到 100 个节点的规模的话，添加节点的最简易方式是使用基本的脚本。

当执行 `docker swarm init` 时，只需要复制粘贴结果的输出行。

```
2. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (c7m85xhn6ue4rx6zbb86t8rnx) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-42mf2wtk9m5pctdqiy5bcfqorj9zlnh5n1z3sszoia7l15jwtd-9o4cok8
ewdszlhh8wc826ivvw \
    192.168.99.100:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-42mf2wtk9m5pctdqiy5bcfqorj9zlnh5n1z3sszoia7l15jwtd-bo1vg5g
ilk7lvjwibtx4x0ik0 \
    192.168.99.100:2377
➜ ~
```

然后，循环创建一定数量的 worker：

```
#!/bin/bash
for i in `seq 0 9`; do
docker-machine create -d amazonec2 --engine-install-url
https://test.docker.com --amazonec2-instance-type "t2.large" swarmworker-$
i
done
```

在这之后，仅需要查看机器列表，ssh 进机器并且 `join` 节点：

```
#!/bin/bash
SWARMWORKER="swarm-worker-"
for machine in `docker-machine ls --format {{.Name}} | grep
$SWARMWORKER`;
do
docker-machine ssh $machine sudo docker swarm join --token SWMTKN-
1-5c3mlb7rqytm0nk795th0z0eocmcmt7i743ybsffad5e04yvxt-
9m54q8xx8m1wa1g68im8srcme \
    172.31.10.250:2377
done
```

该脚本遍历所有机器，每台机器的机器名前缀为 swarm-worker-，脚本会 ssh 进每台机器，并且将该节点加入到已有的Swarm以及领导者管理器上，这里是172.31.10.250。

详细信息见 https://github.com/swarm2k/swarm2k/tree/master/amazon ec2。

belt

belt 是大规模预配 Docker Engine 的另一种方式。它实际上是一个 SSH 封装器，要求用户提前准备好供应商特定的镜像和预配模板。本节介绍具体的操作步骤。

我们可以从 Github 获得 belt 的源码然后自行编译。

```
# Set $GOPATH here
go get https://github.com/chanwit/belt
```

目前，belt 仅仅支持 DigitalOcean 驱动。我们可以在 config.yml 里准备预配所用的模板。

```
digitalocean:
      image: "docker-1.12-rc4"
      region: nyc3
      ssh_key_fingerprint: "your SSH ID"
      ssh_user: root
```

然后仅需一些命令就可以创建上百个节点。

首先，创建三个 16GB 的管理器，名为 mg0、mg1 和 mg2。

```
$ belt create 16gb mg[0:2]
NAME    IPv4             MEMORY  REGION    IMAGE                  STATUS
mg2     104.236.231.136  16384   nyc3      Ubuntu docker-1.12-rc4 active
mg1     45.55.136.207    16384   nyc3      Ubuntu docker-1.12-rc4 active
mg0     45.55.145.205    16384   nyc3      Ubuntu docker-1.12-rc4 active
```

然后，使用 status 命令等待所有节点变成活动状态：

```
$ belt status --wait active=3
```

```
STATUS  #NODES NAMES
Active      3   mg2,  mg1, mg0
```

对 10 个 worker 节点重复上述操作：

```
$ belt create 512mb node[1:10]
$ belt status --wait active=13

STATUS #NODES NAMES
active      3 node10, node9, node8, node7
```

使用 Ansible

我们还可以选择使用 Ansible（笔者很喜欢，这种方式也正在变得相当流行）让所做的事情可以稳定地重复。创建一些 Ansible 模块，直接操作 Machine 和 Swarm（Mode），它和 Docker 1.12 版本（https://github.com/fsoppelsa/ansible-swarm）兼容。要求使用 Ansible 2.2+ 版本，这是能够和二进制模块兼容的第一个 Ansible 版本。

需要编译模块（go 语言编写的），随后将其传递给 `ansible-playbook -M` 参数。

```
git clone https://github.com/fsoppelsa/ansible-swarm
cd ansible-swarm/library
go build docker-machine.go
go build docker_swarm.go
cd ..
```

如下是 playbook 里的一些 play 示例。Ansible 的 play 语法很容易理解，几乎不需要额外的解释。

本书使用如下 play，将 10 个 worker 添加到 Swarm2k 的实验里：

```
---
name: Join the Swarm2k project
  hosts: localhost
  gather_facts: False
  #mg0 104.236.18.183
#mg1 104.236.78.154
#mg2 104.236.87.10
  tasks:
```

```
    name: Load shell variables
    shell: >
      eval $(docker-machine env "{{ machine_name }}")
      echo $DOCKER_TLS_VERIFY &&
      echo $DOCKER_HOST &&
      echo $DOCKER_CERT_PATH &&
      echo $DOCKER_MACHINE_NAME
    register: worker
name: Set facts
    set_fact:
      whost: "{{ worker.stdout_lines[0] }}"
      wcert: "{{ worker.stdout_lines[1] }}"
name: Join a worker to Swarm2k
    docker_swarm:
      role: "worker"
      operation: "join"
      join_url: ["tcp://104.236.78.154:2377"]
      secret: "d0cker_swarm_2k"
      docker_url: "{{ whost }}"
      tls_path: "{{ wcert }}"
    register: swarm_result
name: Print final msg
    debug: msg="{{ swarm_result.msg }}"
```

基本上，在从 Machine 加载了一些主机信息后，它会调用 `docker_swarm` 模块：

- 完成的操作是 `join`。
- 新节点的角色是 `worker`。
- 新节点加入 `tcp://104.236.78.154:2377`，这是加入时的领导者管理器。该参数需要传入管理器数组，比如`[tcp://104.236.78.154:2377, 104.236.18.183:2377, tcp://104.236.87.10:2377]`。
- 它传入密码（`secret`）。
- 它指定一些基本的 engine 连接信息，以及使用 `tlspath` 处的证书连接到 `dockerurl` 的模块。

当 `docker_swarm.go` 在库里编译后，将 worker 加入到 Swarm 就非常简单，如下：

```
#!/bin/bash
SWARMWORKER="swarm-worker-"
for machine in `docker-machine ls --format {{.Name}} | grep
$SWARMWORKER`;
do
ansible-playbook -M library --extra-vars "{machine_name: $machine}"
playbook.yaml
done
```

```
PLAY [Join the Swarm2k project] ************************************************

TASK [Load shell variables] ****************************************************
changed: [localhost]

TASK [Set facts] ***************************************************************
ok: [localhost]

TASK [Join a worker to Swarm2k] ************************************************
ok: [localhost]

TASK [Print final msg] *********************************************************
ok: [localhost] => {
    "msg": "ok join"
}

PLAY RECAP *********************************************************************
localhost                  : ok=4    changed=1    unreachable=0    failed=0
```

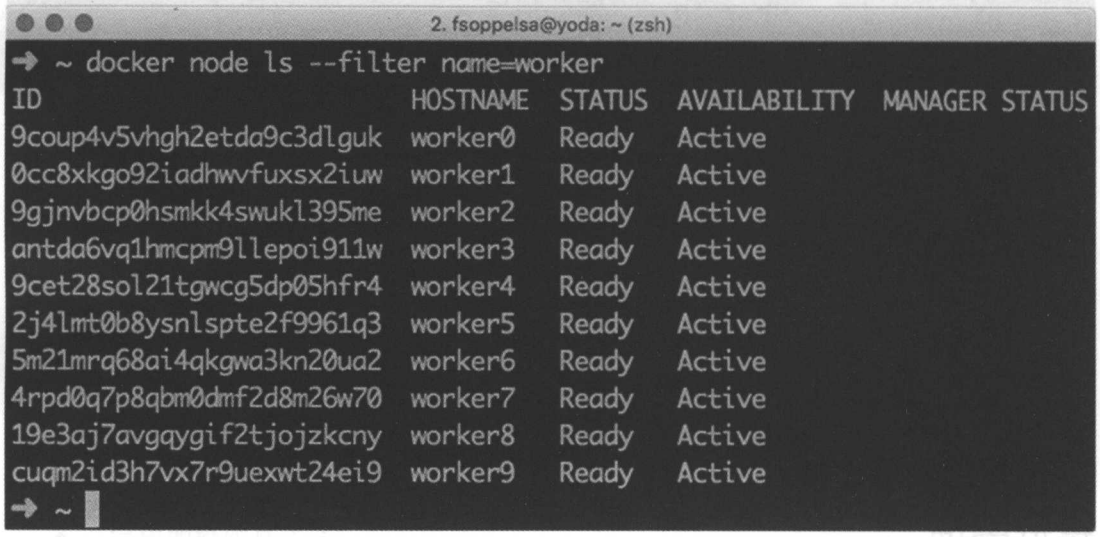
集群管理

为了更好地解释集群运维操作，这里以一个由三个管理器和十个 worker 组成的集群为例。第一个基本的操作是列出节点，使用 `docker node ls` 命令：

```
2. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker node ls --filter name=manager
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
ctv03nq6cjmbkc4v1tc644fsi *  manager1  Ready   Active        Reachable
c61tpogt1hszkjb3sqy1q3rtn    manager2  Ready   Active        Reachable
5v5pw19qi9iqoehh1sfgc5cse    manager3  Ready   Active        Leader
➜ ~
```

通过主机名（manage1）或者 ID（ctv03nq6cjmbkc4v1tc644fsi）来指代节点。上表的其他列描述了集群节点的属性。

- STATUS 指的是节点物理可访问性。如果节点是启动的，显示为 Ready，否则，显示为 Down。
- AVAILABILITY 指的是节点可用性。节点状态可以是 Active（参与到集群运营中）、Pause（待命，挂起，不接受任务）或者 Drain（等待疏散其任务）。
- MANAGER STATUS 指的是管理器的当前状态。如果节点不是管理器，该字段为空。如果节点是管理器，该字段可以是 Reachable（待命保证高可用的管理器之一）或者 Leader（主导所有运营的主机）。



操作节点

`docker node` 命令提供了如下参数。

```
2. fsoppelsa@yoda: ~ (zsh)
➜  ~ docker node --help

Usage:  docker node COMMAND

Manage Docker Swarm nodes

Options:
      --help   Print usage

Commands:
  demote      Demote a node from manager in the swarm
  inspect     Display detailed information on one or more nodes
  ls          List nodes in the swarm
  promote     Promote a node to a manager in the swarm
  rm          Remove a node from the swarm
  ps          List tasks running on a node
  update      Update a node

Run 'docker node COMMAND --help' for more information on a command.
➜  ~
```

如上可见，该命令提供了节点管理相关的所有指令，但是没有 `create`。经常有人问是否会向 `node` 命令添加 create 参数，但是这个问题仍然没有得到官方的回答。

目前，创建新节点是手动操作的，由集群运维人员负责。

降级和升级

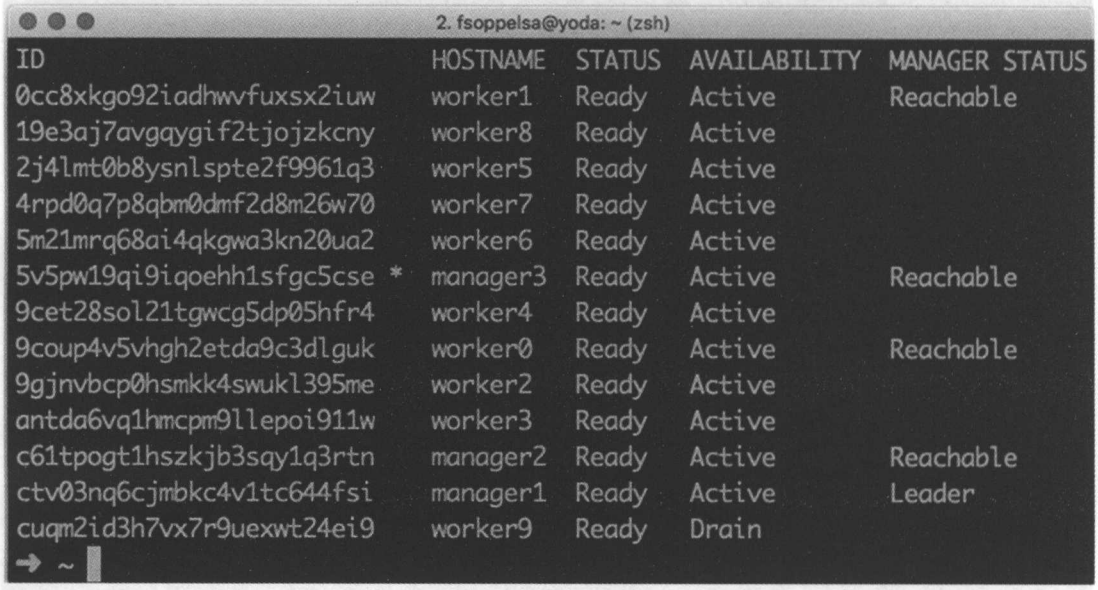
worker 节点可以升级（将它们转变为管理器），而管理器节点可以降级（将它们转变为 worker）。

管理很多管理器和 worker 时，务必保证高可用（管理器个数为奇数，大于或者等于 3）。

在管理器使用如下语法——`promote worker0` 和 `worker1`：

```
docker node promote worker0
docker node promote worker1
```

这里没有什么神奇的功能。Swarm 仅仅尝试将节点角色做相应的变更。



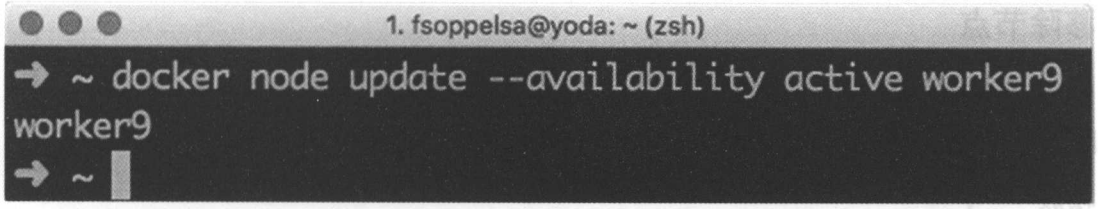
降级是一样的（docker node demote worker1）。但是要小心不要将正在上面工作的节点降级，否则你的管理器就会被锁定。

最后，如果尝试降级领导者管理器会发生什么呢？这时，Raft 算法会启动选举，在 active 的管理器中重新选出新的领导者。

标记节点

你可能注意到之前的图片里，worker9 的可用性为 Drain。这意味着该节点正在疏散其上运行的任务（如果有的话），这些任务会被调度到集群的其他地方。

可以使用 `docker node update` 命令更新节点状态，从而变更节点的可用性：



availability 参数可以是 active、pause 或者 drain。这里将 worker9 恢复到 active 状态。

- active 状态意味着节点正在运行，已经准备接受任务。
- pause 状态意味着节点正在运行，但是不接受任务。
- drain 状态意味着节点正在运行，并且不接受任务，同时它正在疏散其上运行的任务，将其重新调度到其他地方。

update 命令的另一个强大参数是关于标签的。--label-add 和--label-rm 允许用户给 Swarm 节点添加标签。

Docker Swarm 标签并不会影响 Engine 标签。启动 Docker Engine（dockerd [...] --label "staging" --label "dev" [...]）时，可以指定标签。但是 Swarm 没有权利编辑或者修改它们。这里看到的标签仅仅影响 Swarm 的行为。

标签有助于节点的分类。当启动服务时，用户可以使用标签过滤并且决定物理上在哪里启动容器。比如，如果想要将一组使用 SSD 的节点指派给主机 MySQL，可以运行如下命令：

```
docker node update --label-add type=ssd --label-add type=mysql
worker1
docker node update --label-add type=ssd --label-add type=mysql
worker2
docker node update --label-add type=ssd --label-add type=mysql
worker3
```

之后，当使用副本因子（比如 3）启动服务时，可以通过 node.type 过滤，确保在 worker1、worker2 和 worker3 上启动 MySQL 容器：

```
docker service create --replicas 3 --constraint 'node.type ==
mysql' --name mysql-service mysql:5.5.
```

移除节点

节点的移除是特别需要注意的操作。这并不仅仅是让某个节点离开 Swarm，而且要移除其角色以及所运行的任务。

移除 worker

如果某个 worker 的状态为 Down（比如，因为该机器在物理上关闭了），那么它目前没有运行任何任务，因此可以被安全地移除：

```
docker node rm worker9
```

但是，如果某个 worker 的状态为 Ready，那么上述命令就会报错，拒绝移除。节点的可用性（Active，Pause 或者 Drain）并不会产生影响，因为它当时可能还有正在运行任务，或者正在恢复中。

因此，这时运维人员必须手动疏散该节点。也就是说强制其疏散其任务，重新调度这些任务并移动到其他 worker 上：

```
docker node update --availability drain worker9
```

一旦疏散后，可以关闭节点，然后当其状态为 Down 时就可以将其移除。

移除管理器

管理器不能强制移除。在移除某个管理器节点时，必须首先正确地将其降级为 worker，并且疏散其上的任务，然后关闭：

```
docker node demote manager3
docker node update --availability drain manager3
# Node shutdown
docker node rm manager3
```

必须移除某个管理器时，移除之后其他的某个 worker 节点必须被任命为新的管理器，并且完成升级，从而维持奇数个管理器的状态。

使用如下命令完成移除：`docker node rm --force`
参数`--force` 会在任何情况下强制移除该节点。使用该参数必须非常小心，它通常是处理问题节点的最后一招。

Swarm 健康

Swarm 健康本质上依赖于集群里节点的可用性以及管理器的可靠性（奇数个，可用，启动着）。

我们可以使用如下命令列出节点：

```
docker node ls
```

使用`--filter` 参数过滤输出。比如：

```
docker node ls --filter name=manager # prints nodes named *manager*
docker node ls --filter "type=mysql" # prints nodes with a label
type tagged "mysql"
```

要得到某个特定节点的详细信息，可以使用 inspect：

```
docker inspect worker1
```

另外，过滤参数还可以从输出 JSON 里提取特定数据：

```
docker node inspect --format '{{ .Description.Resources }}' worker2
{1000000000 1044140032}
```

输出核数（1 个）以及分配内存的数量（`1044140032`bytes，或 995MB）。

备份集群配置

管理器上的重要数据存储在`/var/lib/docker/swarm` 处。包括如下数据存储：

- 证书在 `certificates/`
- Raft 状态和 Etcd 日志以及快照在 `raft/`
- 任务数据库在 `worker/`

- 其他不太重要的信息，比如当前管理器状态，当前连接套接字等

周期性地备份数据很重要，以防需要恢复数据。

Raft 日志所使用的空间依赖于在集群上启动的任务数量，以及它们状态变更的频繁程度。对于 200000 个容器来说，Raft 日志大概每三个小时会增加 1GB 的硬盘空间。每个任务的日志实体大概需要 5KB。因此，Raft 日志目录（`/var/lib/docker/swarm/raft`）的日志循环策略，必须根据可用的硬盘空间做校准。

灾难恢复

如果某个管理器的 Swarm 目录内容丢失了或者受损了，就要迅速将这个管理器移出集群，使用命令 `docker node remove nodeID`（如果无法删除的话，可以使用`--force` 参数）。

集群管理员不能使用过时的 Swarm 目录启动管理器或者将其加入到集群里。使用过时的 Swarm 目录加入集群会让整个集群的状态变得不一致，因为所有管理器都将尝试同步错误的数据。

移除了目录损坏的管理器之后，需要删除`/var/lib/docker/swarm/raft/wal` 和 `/var/lib/docker/swarm/raft/snap` 目录。只有完成这一步之后管理器才能安全地重新加入集群。

Swarm 的图形化界面

撰写本书时，Swarm mode 还很新，因此已有的 Docker 图形化用户界面还没有支持它或者正在开发中。

Shipyard

Shipyard（https://shipyard-project.com/）对 Swarm（v1）操作的支持很好，现在升级为使用 Swarm mode。撰写本书时（2016 年 8 月），Github 上有 1.12 分支，可以工作。

本书发行时，支持自动化部署的稳定版本可能就已经面世了。介绍文档位于

https://shipyard-project.com/docs/deploy/automated/。

它的操作应该与 SSH 到领导者管理器主机上的操作相似，然后运行一行命令，如下：

```
curl -sSL https://shipyard-project.com/deploy | bash -s
```

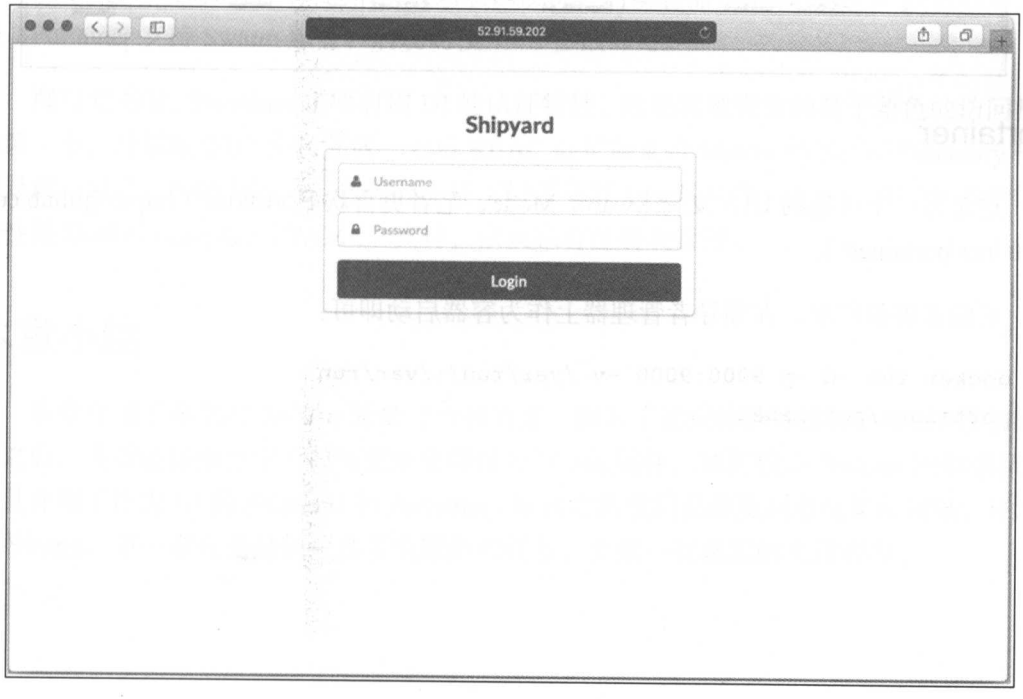
如果仍然需要安装特定的非稳定分支，可以从 Github 上将代码下载到领导者管理器主机上，并且安装 Docker Compose。

```
curl -L
https://github.com/docker/compose/releases/download/1.8.0/dockercompose-`
uname -s`-`uname -m` > /usr/local/bin/docker-compose &&
chmod +x /usr/local/bin/docker-compose
```

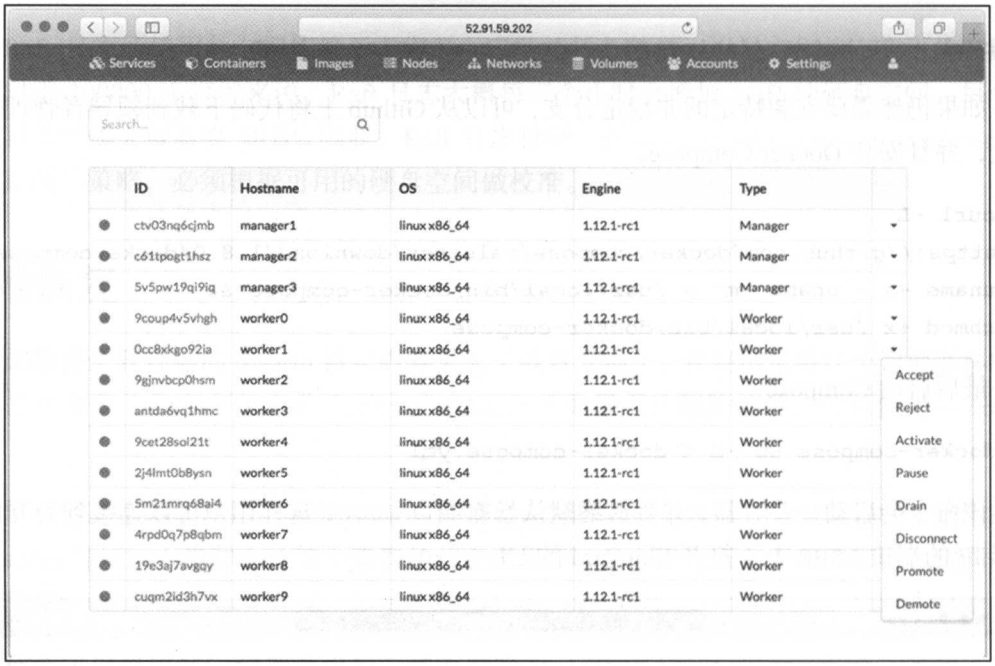
最后启动 Compose：

```
docker-compose up -d < docker-compose.yml
```

该命令会启动一些容器，并且最终默认暴露端口 8080，这样用户可以连接到公开的管理器 IP 的端口 8080 上，打开 Shipyard 的 UI。



如上图所示，UI 上已经支持了 Docker Swarm 特性（有 Service、Node 等）和操作（比如 Promote、Demote 等），这正是本章所介绍的可以对每个节点所做的操作。

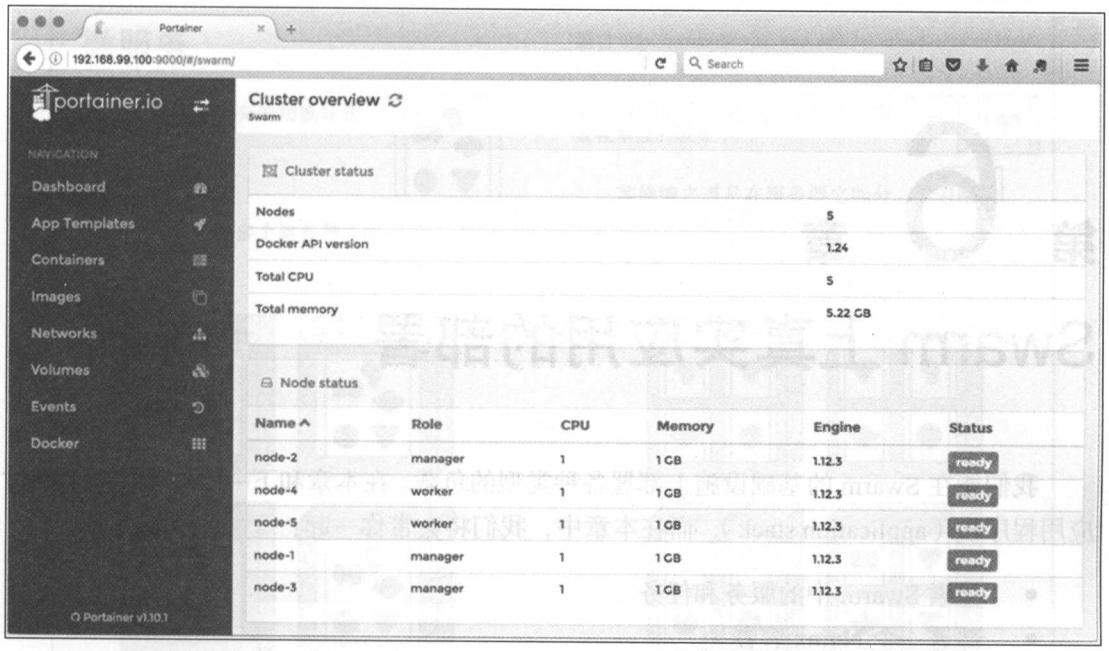


Portainer

这是另一个可选的 UI，支持 Swarm Mode，笔者更喜欢 Portainer（https://github.com/portainer/portainer/）。

它的部署很简单，在领导者管理器上作为容器启动即可：

```
docker run -d -p 9000:9000 -v /var/run/:/var/run
portainer/portainer
```



该 UI 包含所需的选项，包括模板列表以便快速启动容器，比如 MySQL 或者私有注册表，Portainer 支持 Swarm 服务，启动时使用`-s`参数即可。

撰写此书时，Portainer 正要启用 UI 的认证特性，这是实现完整的基于角色的访问控制的第一步，计划在 2017 年初完成。之后 RBAC 会扩展支持 Microsoft Active Directory 作为目录源。另外，2016 年年末，Portainer 还支持多集群（或多主机）管理。2017 年初增加的特性是 Docker Compose（YAML）支持，以及私有注册表管理。

本章小结

本章介绍了典型的 Swarm 管理过程和方案。演示了如何将管理器和 worker 添加到集群之后，本章还详细介绍了如何更新集群以及节点的属性，如何检查 Swarm 的健康状况，并且介绍了作为 UI 的 Shipyard 和 Portainer。在这之后我们会聚焦到基础架构领域，开始使用 Swarm。下一章会通过创建真实的服务和任务，介绍一些真实的应用程序。

第 6 章

Swarm 上真实应用的部署

我们能在 Swarm 的基础设施上部署各种类型的负载。在本章和下一章中我们将会探讨应用程序栈（application stack），而在本章中，我们将会带你一起：

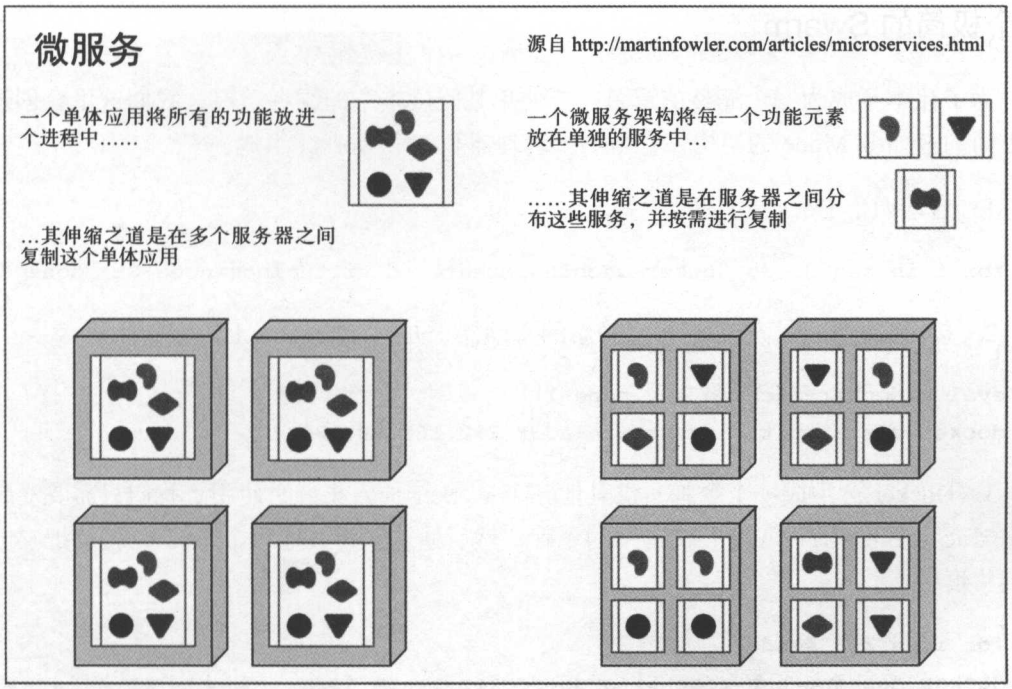
- 探索 Swarm 中的服务和任务
- 部署一个 Nginx 容器
- 部署一个完整的 WordPress
- 部署一个小规模的 Apache Spark 架构

微服务

IT 业一直热衷于各种解耦和重用，不论是对代码还是应用。在架构层次对应用的建模也不例外。在早期，模块化（Modularization）被称作**面向服务的架构**（SOA），并且通过基于 XML 的开源协议连接起来。然而随着容器的出现，微服务成为了新的谈论热点。

微服务是为了完成一个架构目标而一起协作的模块，其特点是体型小、自包含（self-contained）和自我管理（autonomous）。

微服务架构最典型的例子是 Web 应用栈，如 WordPress，它包含的 Web 服务器可以作为一个服务，数据库和缓存引擎也可以分别作为一个服务，其自然也包括用来包含应用自身的服务。用 Docker 容器可以快速地构建微服务模型，这也是目前行业发展的方向。



使用微服务有很多优点：

- **重用性**：如果你对某些服务（如 nginx，MySQL）进行了自定义修改，只需要拉取它们的镜像即可使用。
- **异构性**：尝试各种不同的技术只需链接到已存在的模块即可。如果有一天，你决定把 MySQL 换成 MariaDB，那你只需拔出 MySQL 然后插入 MariaDB 即可。
- **小处着手**（Focus on small）：可拆卸的模块便于单独地进行排错。
- **可伸缩性**：你能轻易地把前端 Web 服务器伸缩到十个，把缓存服务器伸缩到三个，让数据库的副本分布在五个节点上，或者根据应用的负载和需求进行向上或者向下伸缩。
- **耐受性**：如果你有三个 memcached 服务器，然后其中一个宕机了，微服务能让你采取要么尝试修复它，要么不管它并立即启动另一个的机制。

部署一个复制的 Nginx

让我们通过一个简单的例子，即 Nginx 的部署和伸缩，来了解 Swarm 上服务的使用方法。

一个极简的 Swarm

为了让本章成为一个完整的章节，方便单独阅读本章的读者学习，我们这里会创建一个极简的 Swarm Mode 的架构，它由一个管理器和三个 worker 组成。

1. 启动四个 Docker 主机：

```
for i in seq 3; do docker-machine create -d virtualbox node-$i; done
```

2. 接管 `node-1`，选择它作为静态的管理器，并且在 Swarm 上进行初始化：

```
eval $(docker-machine env node-1)
docker swarm init --advertise-addr 192.168.99.100
```

3. Docker 会生成一个令牌，供其他三个 worker 加入集群时使用。我们只需要复制和粘贴 `docker swarm init` 输出中的令牌，然后通过遍历其他三个 worker 就能把它们加入到集群中：

```
for i in 2 3 4; do
docker-machine ssh node-$i sudo docker swarm join \
--token SWMTKN-1-4d1310cf5ipq7e4x5ax2akalds8j1zm6lye8knnb0ba9wftymn-
9odd9z4gfu4d09z2iu0r2361v \
192.168.99.100:2377; done
```

因为前面的 `eval` 命令会填充 Docker Machine 命令输出的环境变量，我们总是会连接到 Swarm Mode 架构中的 `node-1` 节点。我们需要检查所有节点，包括处于领导者（Leader）角色的管理器，是否处于在线状态（Active）并且已经加入了 Swarm。

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
3tqtddj8wfyd1dl92o1l1bniq    node-2    Ready   Active
4sg34cyh6xzuoq2x0fthp6dqp *  node-1    Ready   Active        Leader
73tgqmoq9edw747451bca2asp    node-4    Ready   Active
8sdp6wm2vmfqz2mwzrirwqayb    node-3    Ready   Active
➜ ~
```

我们可以用 `docker info` 命令来检查这个 Swarm 集群的状态：

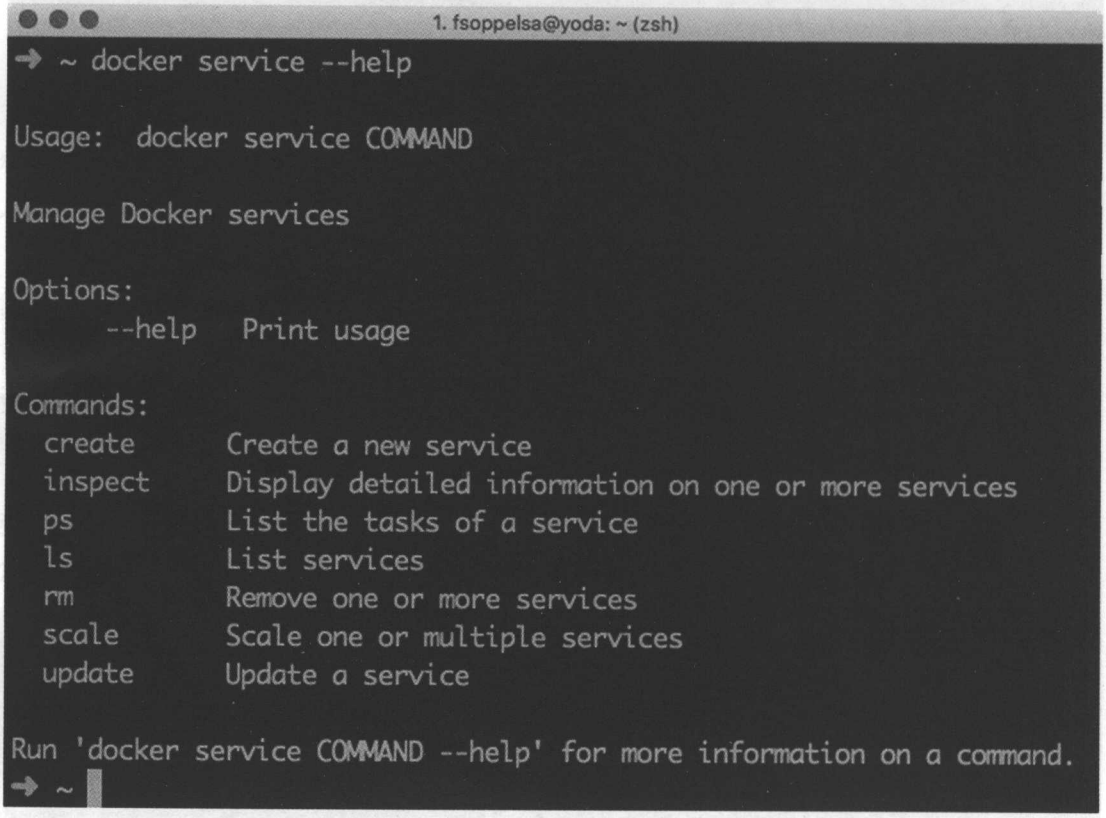
```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 2
Server Version: 1.12.1
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 4
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge null host overlay
Swarm: active
 NodeID: 4sg34cyh6xzuoq2x0fthp6dqp
 Is Manager: true
 ClusterID: 7675xrrjy6lr7nxvf9gdqw3v9
 Managers: 1
 Nodes: 4
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
 Node Address: 192.168.99.100
```

这里比较重要的信息是 Swarm 处于在线状态，后面接着是一些 Raft 的细节信息。

Docker Service

在 Docker 1.12 版本中引入了一个新的命令——`docker service`。在 Docker Swarm 模式下，Service 是用来管理服务的主要方式，它可以用来对服务进行创建、销毁、伸缩和滚动升级。

Service 由 Task 组成（服务由任务组成）。一个 Nginx 服务由 Nginx 任务组成。服务的运作机制是它（通常）会在 worker 节点上启动任务。在创建服务的时候，必须在选项中指定服务的名称和该服务使用的基础容器。



创建服务的语法十分简单，你只需要使用 `docker service create` 命令，并指定好相关选项，如暴露哪些端口，选用什么容器。这里会执行下面这条创建命令：

```
docker service create -p 80:80 --name swarm-nginx --replicas 3
fsoppelsa/swarm-nginx
```

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service create -p 80:80 --name swarm-nginx --replicas 3 fsoppelsa/swarm-nginx
cz7ufryb3ipvnax6ufsor5cvv
➜ ~
```

该命令会启动 Nginx，把容器的端口 80 暴露到主机的端口 80，使其可以被外界访问，然后指定副本因子（replica factor）为 3。

副本因子是 Swarm 中的容器伸缩方式。如果它的值指定为 3，Swarm 就会在三个节点上创建三个 nginx 任务（容器），如果这些容器其中有一个或者多个挂掉，它会通过将 nginx 重新调度到其他可用的主机上（不管什么地方）来努力保持这个数目。

如果--replicas 选项没有指定，那副本因子就是默认值 1。

接着，Swarm 需要从 hub 或者本地的注册表拉取镜像到主机上，然后创建相应的容器并暴露端口；我们可以看到在命令执行完后有三个 nginx 存在。这里我们用到的命令是：

```
docker service ls
```

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ls
ID            NAME          REPLICAS  IMAGE                    COMMAND
cz7ufryb3ipv  swarm-nginx   3/3       fsoppelsa/swarm-nginx
➜ ~
```

通过下面的命令我们看到，这些任务被调度到了三个节点上：

```
docker service ps swarm-nginx
```

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ps swarm-nginx
ID                         NAME            IMAGE                    NODE    DESIRED STATE  CURRENT STATE           ERROR
8gz8x07toy0gpt9yfe7crgvg6  swarm-nginx.1   fsoppelsa/swarm-nginx    node-1  Running        Running 3 minutes ago
8bwdaeecl53uuaejdzdtizhou  swarm-nginx.2   fsoppelsa/swarm-nginx    node-3  Running        Running 3 minutes ago
a7x1fr5ksl6x9m3y4wqzduyh4  swarm-nginx.3   fsoppelsa/swarm-nginx    node-4  Running        Running 3 minutes ago
➜ ~
```

这里使用的是 fsoppelsa/swarm-nginx 容器，它基于一个安装了 PHP 的 nginx 镜像 richarvey/nginx-php-fpm 并做了少量修改。我们使用 PHP 来在 Nginx 的欢迎页上输出当前服务器的地址，并通过它来展示负载均衡机制。

```
<h2>Docker swarm host <?php echo $_SERVER['SERVER_ADDR']; ?></h2>
```

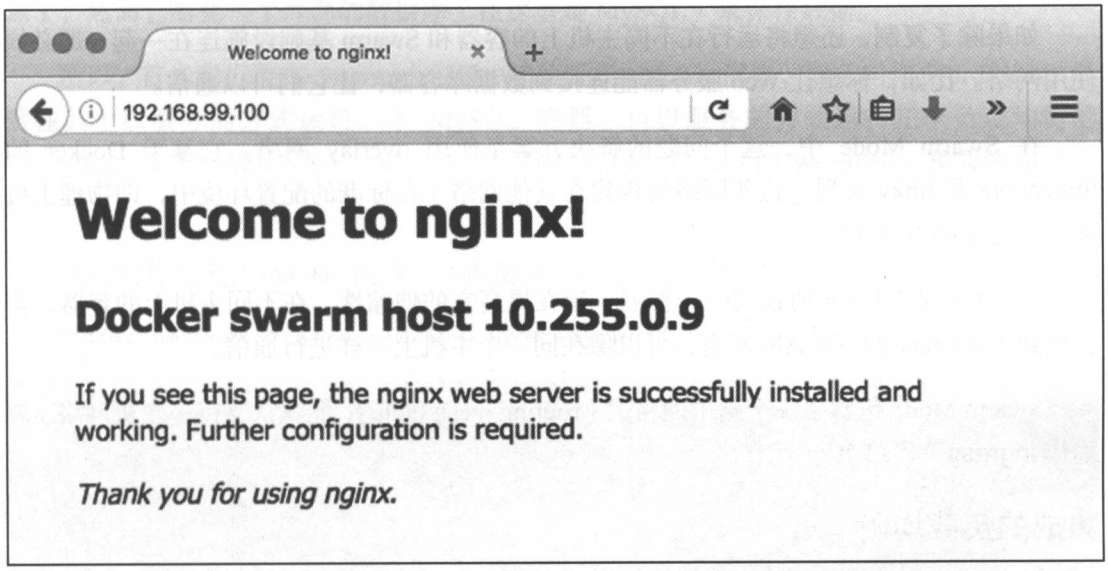
```
1. docker exec -ti c8ca5c1740f2 bash (docker)
➜ ~ d exec -ti c8ca5c1740f2 bash
bash-4.3# cat /var/www/html/index.php

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<h2>Docker swarm host <?php echo $_SERVER['SERVER_ADDR']; ?></h2>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
bash-4.3#
```

现在，如果你通过浏览器访问管理器的 IP 地址并刷新页面，你会发现负载均衡器有时会把你重新定向到不同的容器上。

第一次加载的页面类似下图所示。

下图所示的是另外一次页面加载，负载均衡器选择了一个不同的节点——10.255.0.9。



下图则是刷新页面后负载均衡器重定向到了另外一个不同的节点——10.255.0.10。

overlay 网络

如果除了复制，还想将运行在不同主机上的容器和 Swarm 基础设施连在一起，那必须使用网络。比如，你想让 Web 服务器能连接到数据库容器，让它们可以通信。

在 Swarm Mode 中，这个问题的解决方案是使用 overlay 网络。它基于 Docker 的 libnetwork 和 libkv 实现。这些网络是构建在其他网络（在标准的配置环境中，即物理主机网络）之上的 VxLAN。

VxLAN 是 VLAN 协议的一个扩展，旨在提高它的伸缩性。在不同主机上的容器，若连接到了 Docker 的 VxLAN 网络，可以像在同一个主机上一样进行通信。

Swarm Mode 包含了一个路由网格表（routing mesh table），能默认支持多主机网络，被称作 ingress（入口）。

集成的负载均衡

在 Swarm Mode 1.12 版本中，负载均衡是如何工作的呢？其有两种路由方式：第一种，它通过被虚拟 IP（Virtual IP）服务暴露的端口来工作，任何到该端口的请求都被路由到放置了该服务任务的主机上。第二种，服务会被分配一个只能在 Docker 网络内被路由的虚拟 IP 地址。当有针对该虚拟 IP 地址的请求时，这些请求会被路由到底层的容器中。该虚拟 IP 地址会在 Swarm 自带的 DNS 服务器上注册，在有该服务名称的 DNS 查询时（例如 `nsqlookup mysql`），虚拟 IP 地址会返回。

服务的连接：用 WordPress 例子展示

我们已经能启动很多复制的容器，并拥有负载均衡的支持，这已经是一个很好的开始，如果是更加复杂的包含了不同互相连接容器的应用栈，应该怎么做呢？

对于这种场景，你可以通过用调用它们的名称达到连接的目的。正如我们刚刚看到的，Swarm 内部的 DNS 服务器能确保有一个稳定的名称解析机制。如果初始化了一个叫作 `nginx` 的服务，你可以用 `nginx` 来引用它，其他服务能解析到 `nginx` 服务的虚拟 IP 地址（带有负载均衡的服务），通过这样的方式来访问到分布的容器。

为了用一个例子来展示这一点，我们现在要在 Swarm 上部署一个较老的经典例子：

WordPress。你可以让 WordPress 作为一个容器运行，实际上 Docker Hub 上已经有现成的镜像了，然而它需要一个外部的数据库（在这里是 MySQL）来保存数据。

所以，首先我们要在 Swarm 上创建一个专用的 overlay 网络，取名叫作 WordPress，然后在其上以服务的方式运行一个 MySQL 容器，再以服务的方式运行三个负载均衡的 WordPress 容器（Web 容器）。MySQL 暴露的端口是 3306，而 WordPress 暴露的端口是 80。

让我们首先定义 overlay 网络。在连接到 Swarm 管理器后，输入以下命令：

```
docker network create --driver overlay wordpress
```

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
3tqtddj8wfyd1dl92o1l1bniq    node-2    Ready   Active
4sg34cyh6xzuoq2x0fthp6dqp *  node-1    Ready   Active        Leader
73tgqmoq9edw747451bca2asp    node-4    Ready   Active
8sdp6wm2vmfqz2mwzrirwqayb    node-3    Ready   Active
➜ ~ docker network create --driver overlay wordpress
4xl32a8cc24qq44f6d15rjdv5
➜ ~ docker network ls | grep wordpress
4xl32a8cc24q        wordpress           overlay             swarm

➜ ~
```

那么幕后都发生了什么呢？这个命令使用 libnetwork 创建了一个 Overlay 网络，我们可以在节点接到调度的任务且任务需要使用该网络的时候供其使用。如果你连上 node-2 然后列出所有网络，你也能看到它的存在。

现在创建一个 MySQL 的服务，它仅仅包含一个容器（没有使用 MySQL 原生的复制副本，或者 Galera 等其他复制机制）。使用如下命令：

```
docker service create \
--name mysql \
--replicas 1 \
-p 3306:3306 \
--network wordpress \
--env MYSQL_ROOT_PASSWORD=dockerswarm \
```

```
mysql:5.6
```

在这条命令中，我们想要从 hub 中拉取 MySQL 5.6 镜像，并把该服务命名为 mysql（之后可以通过名称解析指向它的 VIP），并明确地把副本数设置为 1，暴露端口 3306，指定名为 WordPress 的专用的网络，并指定 root 用户的密码，我们这里使用的是 dockerswarm。

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service create \
> --name mysql \
> --replicas 1 \
> -p 3306:3306 \
> --network wordpress \
> --env MYSQL_ROOT_PASSWORD=dockerswarm \
> mysql:5.6
bhzvq7tl2p85p409qp67r0i81
➜ ~
```

它会执行必要的操作，从 hub 中拉取 MySQL 镜像，过了几秒钟之后我们就可以检查并看到，它下载了一个 mysql 容器并被放置在了 node-1 上（实际上，除非明确指明，master 也可以用来运行容器），并且 VIP 是 10.255.0.2，位于 WordPress 网络中。我们可以用如下的命令来获取到这个信息：

```
docker service inspect mysql -f "{{ .Endpoint.VirtualIPs }}"
```

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ls
ID            NAME   REPLICAS  IMAGE      COMMAND
agqdoyh723fj  mysql  1/1       mysql:5.6
➜ ~ docker service ps mysql
ID                         NAME     IMAGE      NODE    DESIRED STATE  CURRENT STATE           ERROR
5c28bwm3detyz7eiq77zb3vca  mysql.1  mysql:5.6  node-1  Running        Running 17 seconds ago
➜ ~ d service inspect mysql -f "{{ .Endpoint.VirtualIPs }}"
[{6ha06jgn7zudgn410i5zqk0az 10.255.0.2/16} {4xl32a8cc24qq44f6d15rjdv5 10.0.0.2/24}]
➜ ~
```

现在我们有了一个运行的 MySQL，只需要启动并连接到 WordPress。

Swarm 的调度策略

我们启动了一个服务，然后 Swarm 把容器凑巧调度到了 node-1 去运行。Swarm Mode（目前，在撰写本书的时候是 Docker 1.12 和 1.13-dev 版本）只有一个可能的策略：分散（spread）。这种策略会对每一个主机上的容器进行计数，然后试着将新创建的容器放到负载较小的主机（即那些拥有容器数目较少的主机）上。尽管目前只有一种策略，Swarm 还拥有一些可以让我们对主机进行过滤的选项，方便较高精度地控制需要在哪些主机上启动任务。

这些选项被称作约束（constraints），可以作为一个可选参数在服务初始化的时候用--constraint 进行指定。

现在，我们将要启动 WordPress。我们决定强制地将三个容器在三个 worker 上执行，不允许在 master 上执行，我们可以指定一个 constraint。

约束可以通过--constraintnode.KEY == VALUE 或者 constraintnode.KEY != VALUE 的形式进行指定，并且它还有几种变体。操作符可以用来根据节点 id、角色和主机名来过滤节点。更有意思的是，正如我们在第 5 章“管理 Swarm 集群”中看到的，我们还可以指定自定义标签，可以通过 docker node update --label-add 命令来在节点的属性中添加标签。

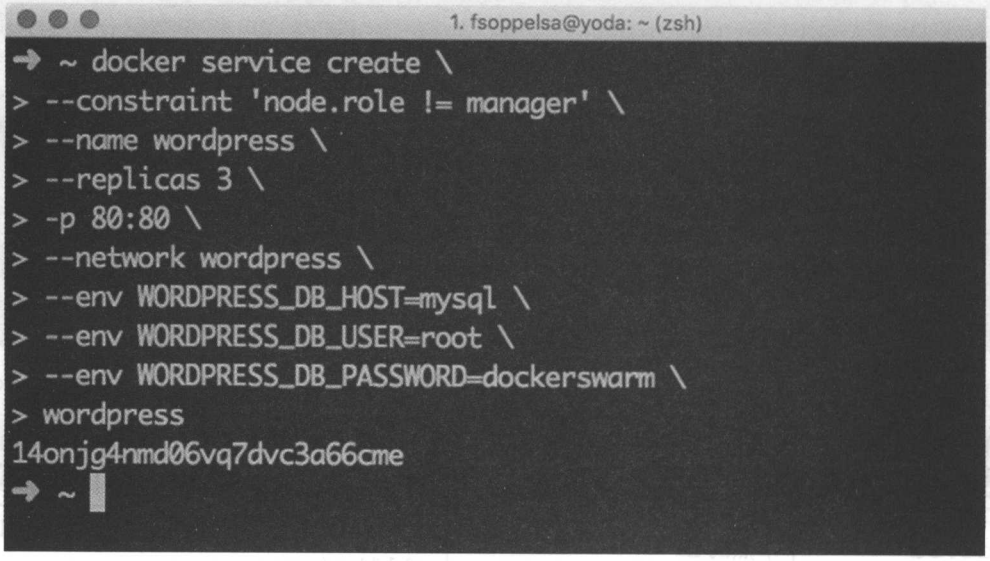
Key	含　义	举　例
node.id	节点的 ID	node.id == 3tqtddj8wfyd1dl92o111bniq
node.role	节点的角色（管理器, worker）	node.role != manager
node.hostname	节点的主机名	node.hostname == node-1
node.labels	标签	node.labels.type == database

现在，WordPress

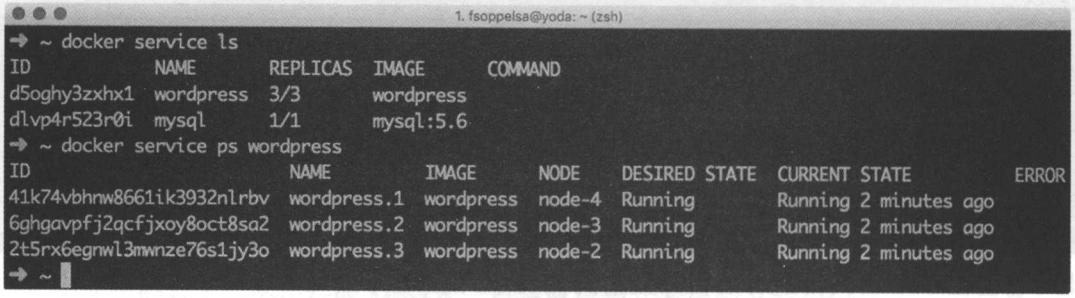
这里我们想在所有的 worker 上启动 wordpress，为此使用的约束是 node.role != manager（或者 node.role == worker）。同时我们将服务简单地命名为 wordpress，将复制因子设置为 3，然后暴露端口 80，并且告诉 WordPress， MySQL 位于主机 mysql 上（它会在 Swarm 内部进行解析并且指向 MySQL 的 VIP）：

```
docker service create \
```

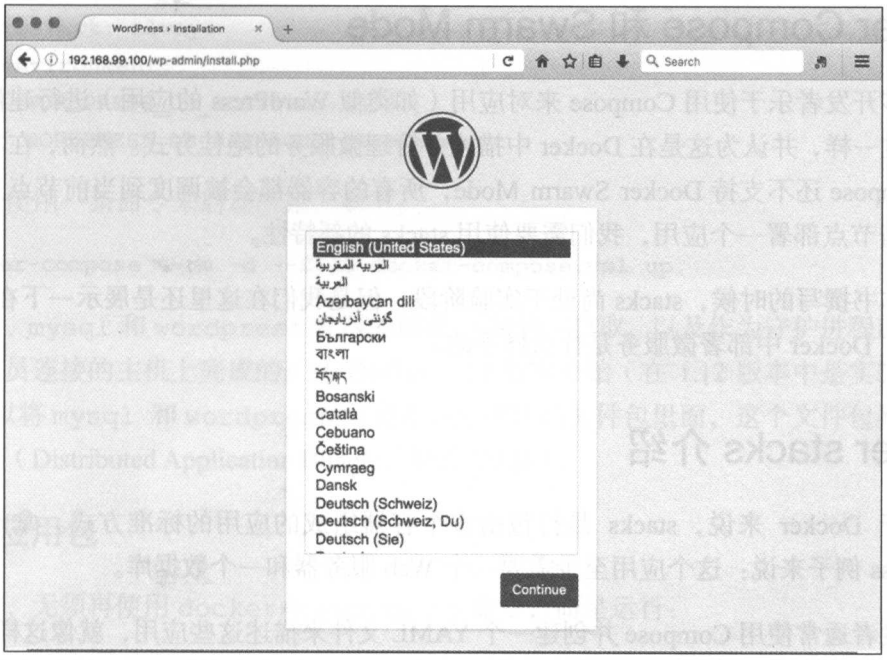
```
--constraint 'node.role != manager' \
--name wordpress \
--replicas 3 \
-p 80:80 \
--network wordpress \
--env WORDPRESS_DB_HOST=mysql \
--env WORDPRESS_DB_USER=root \
--env WORDPRESS_DB_PASSWORD=dockerswarm \
wordpress
```



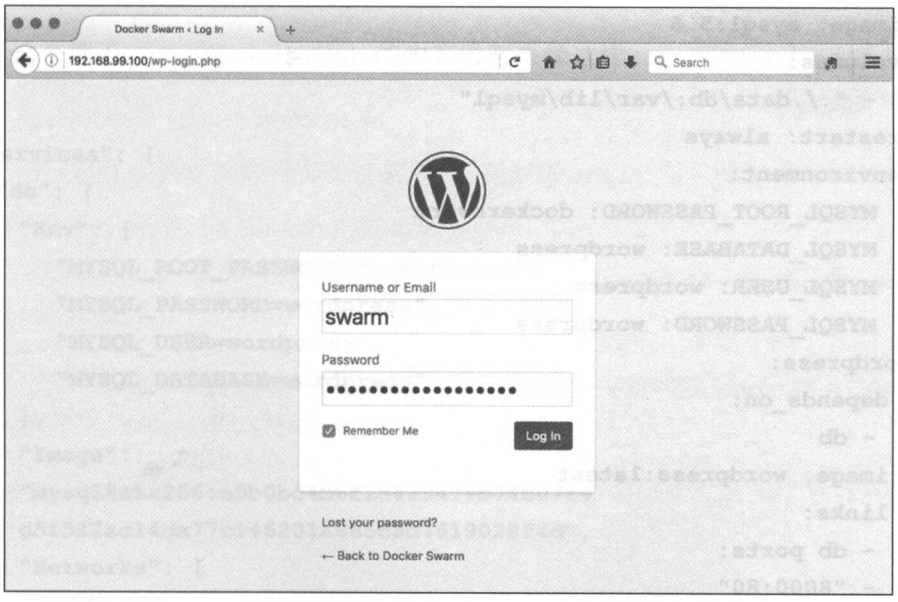
WordPress 的镜像下载到 worker 需要一段时间，下载完成后我们就可以检查一切是否正常。



然后，当我们连接到一个主机的端口 80 时，就可以看到 WordPress 的安装页面。



只需要通过浏览器简单几步的设置，包括设置管理员的用户名和密码，就能很快安装好 WordPress。



Docker Compose 和 Swarm Mode

很多开发者乐于使用 Compose 来对应用（如类似 WordPress 的应用）进行建模。我们的做法也一样，并认为这是在 Docker 中描述并管理微服务的绝佳方式。然而，在本书撰写时，Compose 还不支持 Docker Swarm Mode，所有的容器都会被调度到当前节点上。要在 Swarm 跨节点部署一个应用，我们需要使用 stacks 的新特性。

在本书撰写的时候，stacks 尚处于实验阶段，但是我们在这里还是展示一下在不久的将来，在 Docker 中部署微服务是什么样子的。

Docker stacks 介绍

对于 Docker 来说，stacks 是打包由多个容器组成的应用的标准方式。拿前面这个 WordPress 例子来说：这个应用至少需要一个 Web 服务器和一个数据库。

开发者通常使用 Compose 并创建一个 YAML 文件来描述这些应用，就像这样：

```
version: '2'
services:
  db:
    image: mysql:5.6
    volumes:
      - "./.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: dockerswarm
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
   wordpress:
     depends_on:
      - db
     image: wordpress:latest
     links:
      - db ports:
      - "8000:80"
```

```
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
```

然后使用一条命令来启动应用，如下：

```
docker-compose --rm -d --file docker-compose.yml up.
```

这里，mysql 和 wordpress 容器的调度、（镜像）拉取，以及作为守护进程启动都是在开发人员连接的主机上完成的。从 Docker 1.12 版本开始（在 1.12 版本中是实验性的特性），可以将 mysql 和 wordpress 打包在一个单独的文件包里面，这个文件包称为分布式应用包（Distributed Application Bundle，简称 DAB）。

分布式应用包

这样，无须再使用 docker-compose up 命令，而是运行：

```
docker-compose --file docker-compose.yml bundle -o wordpress.dab
```

这个命令会输出另外一个叫作 wordpress.dab 的 JSON 文件，它是用来部署被 Swarm service 描述的服务的起点。

对于这个例子，wordpress.dab 的内容看起来像这样：

```
{
  "Services": {
    "db": {
      "Env": [
        "MYSQL_ROOT_PASSWORD=dockerswarm",
        "MYSQL_PASSWORD=wordpress",
        "MYSQL_USER=wordpress",
        "MYSQL_DATABASE=wordpress"
      ],
      "Image":
      "mysql@sha256:e9b0bc4b8f18429479b74b07f4
      d515f2ac14da77c146201a885c5d7619028f4d",
      "Networks": [
        "default"
```

```
        ]
      },
      "wordpress": {
       "Env": [
         "WORDPRESS_DB_HOST=db:3306",
         "WORDPRESS_DB_PASSWORD=wordpress"
       ],
       "Image":
       "wordpress@sha256:10f68e4f1f13655b15a5d0415
       3fe0a454ea5e14bcb38b0695f0b9e3e920a1c97",
       "Networks": [
         "default"
       ],
       "Ports": [
         {
           "Port": 80,
           "Protocol": "tcp"
         }
       ]
     }
   },
   "Version": "0.1"
```

Docker deploy

开发者可以连接到一个 Swarm 管理器，用部署命令从生成的 `wordpress.dab` 文件启动一个应用栈：

```
docker deploy --file wordpress.dab wordpress1
```

现在你有了两个服务，分别叫作 `wordpress1_wordpress` 和 `wordpress1_db`，其语法和传统的 Compose 是保持一致的。

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ls
ID            NAME       REPLICAS  IMAGE      COMMAND
d5oghy3zxhx1  wordpress  3/3       wordpress
dlvp4r523r0i  mysql      1/1       mysql:5.6
➜ ~ docker service ps wordpress
ID                         NAME         IMAGE      NODE    DESIRED STATE  CURRENT STATE          ERROR
41k74vbhnw8661ik3932nlrbv  wordpress.1  wordpress  node-4  Running        Running 2 minutes ago
6ghgavpfj2qcfjxoy8oct8sa2  wordpress.2  wordpress  node-3  Running        Running 2 minutes ago
2t5rx6egnwl3mwnze76s1jy3o  wordpress.3  wordpress  node-2  Running        Running 2 minutes ago
➜ ~
```

这个非常原始的 demo 展示了它未来会是什么样子。作为一个实验性的特性，在 Compose 中支持的特性尚未完全定义，但是我们期待它在未来会更新（甚至是根本性的更新）并满足开发者、Swarm 和 Compose 的需求。

另外一个应用：Apache Spark

现在我们已经完成了一些使用 service 的实践，让我们再深入一些。我们将会在 Swarm 上部署 Spark。Spark 是一个开源的集群计算框架，来自于 Apache 基金会，主要用于数据处理。

Spark 可以用于但不限于以下领域：

- 大数据分析（Spark Core）
- 高速、可伸缩的结构化数据终端（Spark SQL）
- 流式化分析（Spark Streaming）
- 图像处理（Spark GraphX）

这里我们将会集中讨论 Swarm 的基础设施部分。如果你想要学习 Spark 的编程或使用，可以参阅 Packt 的关于 Spark 的书籍。我们建议可以从 *Fast Data Processing with Spark 2.0 – Third Edition* 开始。

Spark 是一个可以替换 Hadoop 的精益方案，和 Hadoop 的繁杂庞大相比，它更加敏捷和高效，能取代 Hadoop 中的复杂和庞大。

Spark 理论上的拓扑很简单，它和 Swarm 模式相似，由一个或者多个管理者来领导集群的操作，然后由一定数量的工作者来执行实际的任务。

至于管理者，Spark 可以使用它自己的管理者，叫作单独的管理者（我们这里采用这种

说法），或者使用 Hadoop YARN，甚至可以利用 Mesos 的特性。

然后，Spark 可以将存储指派给内部的 HDFS（Hadoop 分布式文件系统）或者外部的文件存储服务，如 Amazon S3、OpenStack Swift 或者 Cassandra。存储被 Spark 用来获取进行细化的数据，并保存细化后的结果。

为什么要在 Docker 上运行 Spark

我们将展示如何在 Docker Swarm 集群上启动一个 Spark 集群，这可以用来取代在虚拟机上启动 Spark 的方式。这里的示例可以从容器中获得很多的好处：

- 容器启动速度更快。
- 在 pet 模式下容器能更加快速地伸缩。
- 获取 Spark 的镜像无须创建虚拟机、编写自定义脚本或者改写 Ansible Playbook，只需要 `docker pull`。
- 你可以使用 Docker 的网络特性来创建一个专用的 overlay 网络，没有物理的限制，无须求助网络团队。

没有 Swarm 的 Spark 单机

让我们首先定义一个小的、使用传统 Docker 工具（即 Docker 主机上的 Docker 命令）构建的 Apache Spark 集群。在了解整个体系之前，我们先要熟悉一些 Swarm 的概念和定义。

在本章中，我们将使用 `google_container` 镜像，并使用 Swarm 1.5.2 版本。2.0 版本有很多升级，但是这些镜像已经被证实十分稳定和可靠。让我们从 Google 的仓库中拉取，然后使用其提供的（Spark） master 和 worker。

```
docker pull gcr.io/google_containers/spark-master
docker pull gcr.io/google_containers/spark-worker
```

Spark 可以运行在 YARN、Mesos 或者 Hadoop 上。在下面的例子和内容中，我们将会使用其单节点模式，因为它最简单，不需要额外的依赖。在单节点的 Spark 集群模式下，Spark 基于核心数量来分配资源。在默认情况下，应用会使用集群中所有的核心，所以我们将要对 worker 的资源进行限制。

我们的架构也十分简单：一个 master 节点用来管理集群，三个 worker 节点用来运行

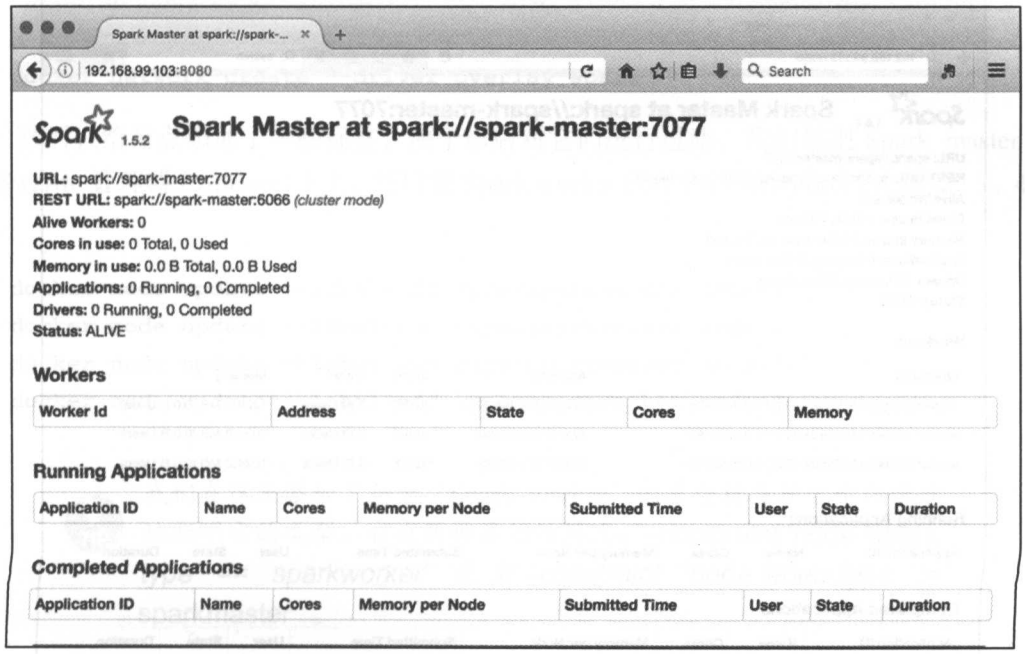
Spark 的作业。在这个例子中，必须要在 master 上打开端口 8080（按照 Web 界面惯例），然后我们把它称作 spark-master。在默认情况下，worker 容器会尝试连接 URL spark://spark-master:7077，除了把它们和 master 连接，不需要其他自定义操作。

我们来初始化一个 Spark master：

```
docker run -d \
-p 8080:8080 \
--name spark-master \
-h spark-master \
gcr.io/google_containers/spark-master
```

它运行在一个来自 gcr.io/google_containers/spark-master 镜像的容器中，以守护进程模式（-d 选项）运行，指定的名称（--name 选项）是 spark-master，然后主机名配置（-h 选项）为 spark-master。

我们现在可以通过浏览器去访问 Docker 主机的端口 8080 来验证 Spark 是否运行。



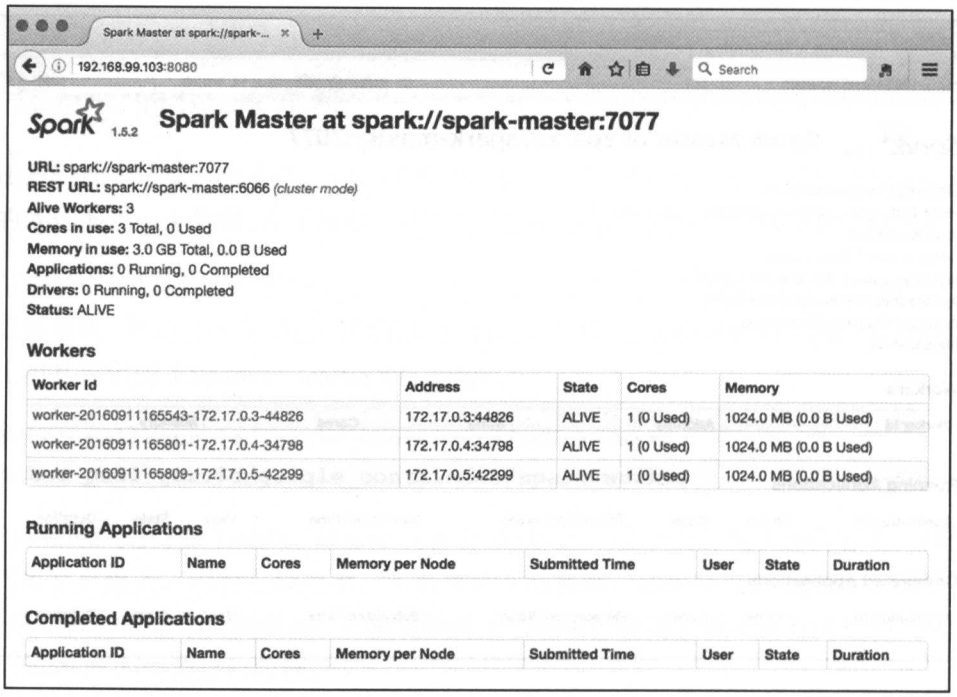
它目前尚没有活跃的 Worker（Alive Worker），我们即将启动它。可以用下面的命令来启动 worker：

```
docker run -d \
--link 7ff683727bbf \
-m 256 \
-p 8081:8081 \
--name worker-1 \
gcr.io/google_containers/spark-worker
```

这条命令会以守护进程的模式启动一个容器，并把它连接到 master，它能使用的最大内存限制为 256MB，暴露端口 8081 用于 Web 管理，并且指定容器的名称为 worker-1。我们可以用类似的方式启动其余两个 worker：

```
docker run -d --link d3409a18fdc0 -m 256 -p 8082:8082 -m 256m --
name worker-2 gcr.io/google_containers/spark-worker
docker run -d --link d3409a18fdc0 -m 256 -p 8083:8083 -m 256m --
name worker-3 gcr.io/google_containers/spark-worker
```

我们可以在 master 上检查 worker 是否连接正常并是否处于运行状态：



在 Swarm 上的 Spark 单机

到目前为止，我们已经讨论完不那么重要的部分。现在将会从已经讨论过的概念过渡到 Swarm 的架构部分，所以我们将把 Spark 的 master 和 worker 实例化成 Swarm 服务，而不是单独的容器。在本书架构中，master 的副本因子为 1，worker 的副本因子为 3。

Spark 拓扑

在这个例子中，我们将会创建一个由一个 master 和三个 worker 组成的 Spark 集群。

存储

我们将会定义一个真正的存储并且在第 7 章“平台的向上伸缩”中开始一些真正的 Spark 任务。

预先准备

首先，为 Spark 创建一个专用的 overlay 网络：

```
docker network create --driver overlay spark
```

然后，给节点添加了一些标签，便于以后对它们进行过滤，我们想把 Spark master 放在 Swarm 管理器（node-1）上，然后将 Spark worker 放在 Swarm worker（node-2，3，4）上：

```
docker node update --label-add type=sparkmaster node-1
docker node update --label-add type=sparkworker node-2
docker node update --label-add type=sparkworker node-3
docker node update --label-add type=sparkworker node-4
```

我们在这里添加了一个“Spark worker”的类型标签用于十分清楚地指明它的类型。它有两种其他的写法：–constraint 'node.labels.type == sparkworker' 或者 –constraint 'node.labels.type != sparkmaster'。

在 Swarm 上启动 Spark

现在我们将在 Swarm 中定义 Spark 服务，与在前面定义 WordPress 的做法一样，但是这一次我们将会深入地使用调度策略，将会尽量精确地指定 Spark master 和 Spark worker 要启动的位置。

我们用如下的方式来启动 master。

```
docker service create \
--container-label spark-master \
--network spark \
--constraint 'node.labels.type==sparkmaster' \
--publish 8080:8080 \
--publish 7077:7077 \
--publish 6066:6066 \
--name spark-master \
--replicas 1 \
--limit-memory 1024 \
gcr.io/google_containers/spark-master
```

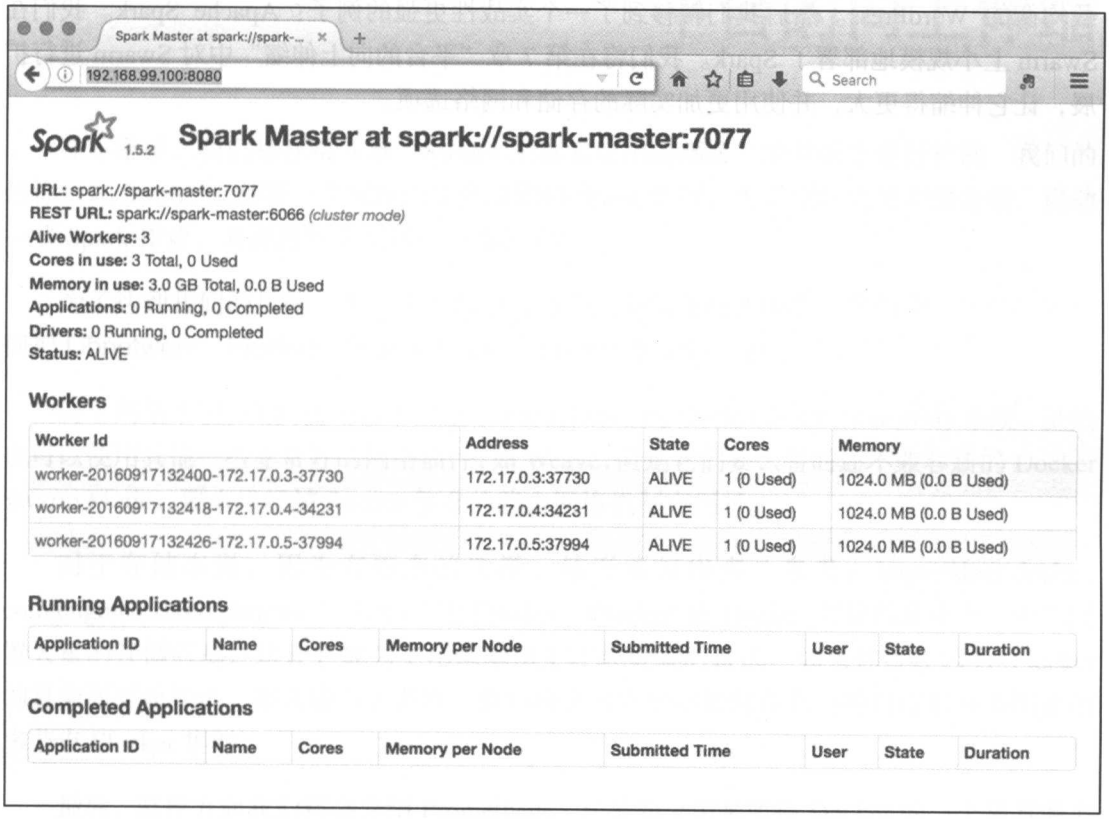
这个 Spark 的 master 会暴露端口 8080（用于 Web 界面），并且为了让示例清楚明了，同时我们暴露端口 7077，用于 Spark worker 连接 master 和供 Spark API 使用的端口 6066。同时通过--limit-memory 来把内存限制为 1GB。一旦 Spark master 启动好，我们就可以开始创建用于放置 worker 的服务，即 sparkworker：

```
docker service create \
--constraint 'node.labels.type==sparkworker' \
--network spark \
--name spark-worker \
--publish 8081:8081 \
--replicas 3 \
--limit-memory 256 \
gcr.io/google_containers/spark-worker
```

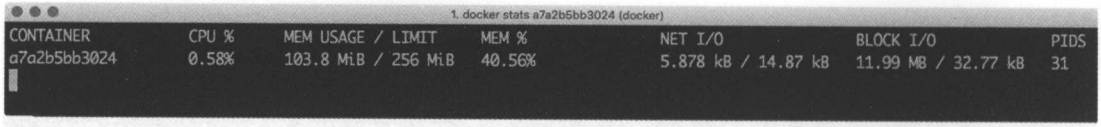
类似的，我们暴露了端口 8081，但是这是可选的。正如在之前定义好的，这里所有的 Spark 容器都在 Spark 的 worker 节点上调度。镜像拉取到主机上会花费一些时间，然后我们很快就有了一个最小化的 Spark 基础设施：

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ls
ID            NAME          REPLICAS  IMAGE                                  COMMAND
ce5sqqlesm7u  spark-worker  3/3       gcr.io/google_containers/spark-worker
dhllxboaybxw  spark-master  1/1       gcr.io/google_containers/spark-master
➜ ~ docker service ps spark-master
ID                         NAME             IMAGE                                  NODE    DESIRED STATE  CURRENT STATE           ERROR
02dq9mkyc1atmlgmk91pj7sya  spark-master.1   gcr.io/google_containers/spark-master  node-1  Running        Running 12 minutes ago
➜ ~ docker service ps spark-worker
ID                         NAME             IMAGE                                  NODE    DESIRED STATE  CURRENT STATE                ERROR
a8wkyw3jnblcumy1mgdo7x6at  spark-worker.1   gcr.io/google_containers/spark-worker  node-2  Running        Running about a minute ago
cx9qewlnclh5wmyptd6benn7z  spark-worker.2   gcr.io/google_containers/spark-worker  node-4  Running        Running 2 minutes ago
5wzv73eac555fked3sxse1h54  spark-worker.3   gcr.io/google_containers/spark-worker  node-3  Running        Running about a minute ago
➜ ~
```

Spark 集群启动并运行起来了，这里有几处值得说明的地方：



尽管我们想把内存限制为 256MB，在 UI 中仍然看到 Spark 显示的是 1024MB，这是由于 Spark 内部的默认配置的缘故。如果我们连上任何一台主机（有任何一个 worker 运行的地方），使用 `docker stats a7a2b5bb3024` 检查它的统计指标，可以看到容器其实是有受到限制的。



本章小结

在本章中，我们开始和应用栈打交道，并在 Swarm 上部署了真实的应用。我们动手实践了如何定义 Swarm 服务，启动一个 nginx 集群，同时在专用的 overlay 网络上运行一个负载均衡的 WordPress。然后我们转移到了一个实战性更强的例子：Apache Spark。我们在 Swarm 上小规模地部署了 Spark。我们将在第 7 章“平台的向上伸缩”中对 Swarm 进行扩展，让它伸缩得更大，并使用更加实际的存储和网络选项。

第 7 章

平台的向上伸缩

在本章中，我们将在第 6 章“Swarm 上真实应用的部署”的基础上进行扩展。我们的目标是在 Swarm 上部署一个实际的生产级别的 Spark 集群，然后增加它的存储容量，启动一些 Spark 作业，并设置好底层基础设施的监控。

为了达到我们的目标，本章绝大部分内容都以基础设施为中心。我们会一起来看看如何把 Libnetwork、Flocker、Prometheus 和 Swarm 整合到一起。

对于网络来说，我们使用基本的基于 Libnetwork 的 Docker Network overlay 系统。虽然也可以使用其他一些非常好的网络插件，如 Weave，但是它们要么暂时还不兼容新的 Docker Swarm Mode，要么已经被 Swarm 整合的路由网格机制弃用了。

对于存储来说，因为有更多的选择，情形要好很多（参考：https://docs.docker.com/engine/extend/plugins/）。我们选用 Flocker。Flocker 是 Docker 存储的最前身，可以配置大量的存储后端，这让它成为了用来承担生产级负载的首选。如果你曾经因为 Flocker 的复杂而望而却步，那么请不必害怕，我们将会向你展示如何在几分钟内设置好多用途的多节点 Flocker 集群。

最后，监控方面我们将会介绍 Prometheus。它是如今用来监控 Docker 的一个最有潜力的监控系统，并且它的 API 可能很快会被集成进 Docker 引擎中。

下面是我们要讨论的内容：

- 一个在 Swarm 上运行 Spark 的例子，可以用来运行任何的 Spark 作业
- 将基础设施上大规模的 Flocker 安装程序进行自动化

- 展示如何在本地使用 Flocker
- 如何和 Swarm Mode 一起使用 Flocker
- 伸缩 Spark 应用
- 用 Prometheus 监控基础设施的健康状况

再次登场的 Spark 例子

本节将会对第 6 章“在 Swarm 上部署真实的应用”中的例子进行重构，这一次使用更加真实的网络和存储设置。

Spark 的存储后端通常在 Hadoop 上，如果使用的是 NFS，则是在文件系统上。对于那些不需要存储的作业来说，Spark 将会在 worker 本地创建数据，但是对于需要存储支持的计算来说，每个节点都需要共享的文件系统，而 Docker 的卷插件不能自动地保证这一点（至少目前是这样的情况）。

在 Swarm 上要达到这个目标的一个可用方案是在每一个 Docker 主机上创建一个 NFS 的共享（share），然后在服务容器里面透明地加载。

我们这里的重点不在于 Spark 作业的相关细节，而是要展示一种我们认为比较好的 Docker 存储选项，然后让大家对在 Swarm 上如何组织并且伸缩一个具有一定复杂程度的服务有一定的概念。

Docker 插件

对于 Docker 插件的介绍，我们建议你阅读官方的文档，可以从这一篇开始：https://docs.docker.com/engine/extend/。同时 Docker 可能会发布一个工具来实现通过一条命令获取插件，这一部分的内容可以参考：https://docs.docker.com/engine/reference/commandline/plugin_install/。

如果你想了解如何将新的特性整合进 Docker 的话，我们推荐参考 *Extending Docker* 这本书。这本书重点讲述了 Docker 的插件，包括卷插件、网络插件，以及如何打造自己的插件。

对于 Flocker 来说，ClusterHQ 发布了一个自动部署工具，可以用来在 AWS 上使用 CloudForm 模板自动部署一个 Flocker 集群。这些模板可以通过 Volume Hub 来安装。要注册并且启动一个这样的集群，可以访问 https://flocker-docs.clusterhq.com/en/latest/docker-integration/cloudformation.html。如果想要了解整个流程的每一个步骤细节，可以参考 Packt 出版的 *Extending Docker* 的第 3 章内容。

实验室环境

在这个教程中，我们将会在 AWS 中创建一个基础设施。对于一个生产的系统，理想的状态是设置三个或者五个 Swarm 管理器，并设置一定数量的 worker，之后再根据负载情况来添加更多的 worker 节点。

这里我们将用 Machine 设置好一个包含三个 Swarm 管理器、六个 Swarm worker 和一个 Flocker 控制节点的 Swarm 集群，然后不再添加新的 worker。

安装 Flocker 需要很多手动的步骤，而它们可以被自动化（我们将会看到如何做到）。为了让我们的例子尽可能不变得更复杂，我们将会在一开始采用线性顺序运行所有的这些命令，不使用重复的流程来增加系统的容量。

如果你不喜欢 Ansible，也可以使用其他你喜欢的工具，如 Puppet、Salt、Chef 或者其他工具来改写下面出现的例子。

一个独一无二的秘钥

为了简单，我们将会通过随机生成的 SSH key 来安装我们的实验环境，然后将它复制到主机上的 `anthorized_keys` 文件中，并用这个秘钥来安装 Docker Machine。这里的目地是在稍后可以有一个唯一的秘钥来认证 Ansible，并用它来自动化很多本应手动执行的步骤。

让我们先生成一个 `flocker` 秘钥，并把它放在 `keys/`目录下

```
ssh-keygen -t rsa -f keys/flocker
```

Docker Machine

要配置好 Docker 主机，我们需要使用 Docker Machine。这里是一些系统细节：

AWS 实例的命名将会从 aws-101 开始一直到 aws-110 结束。这种命名标准在我们为 Flocker 生成和创建节点证书的时候会很重要。

- 节点 aws-101，102，103 将作为 Swarm 的 Manager
- 节点 aws-104 将会是 Flocker 的控制节点
- 从 aws-105 到 aws-110 将会是 Swarm 的 worker

实例的类型是 `t2.medium`（2 vCPU，4GB 内存，EBS 存储）。

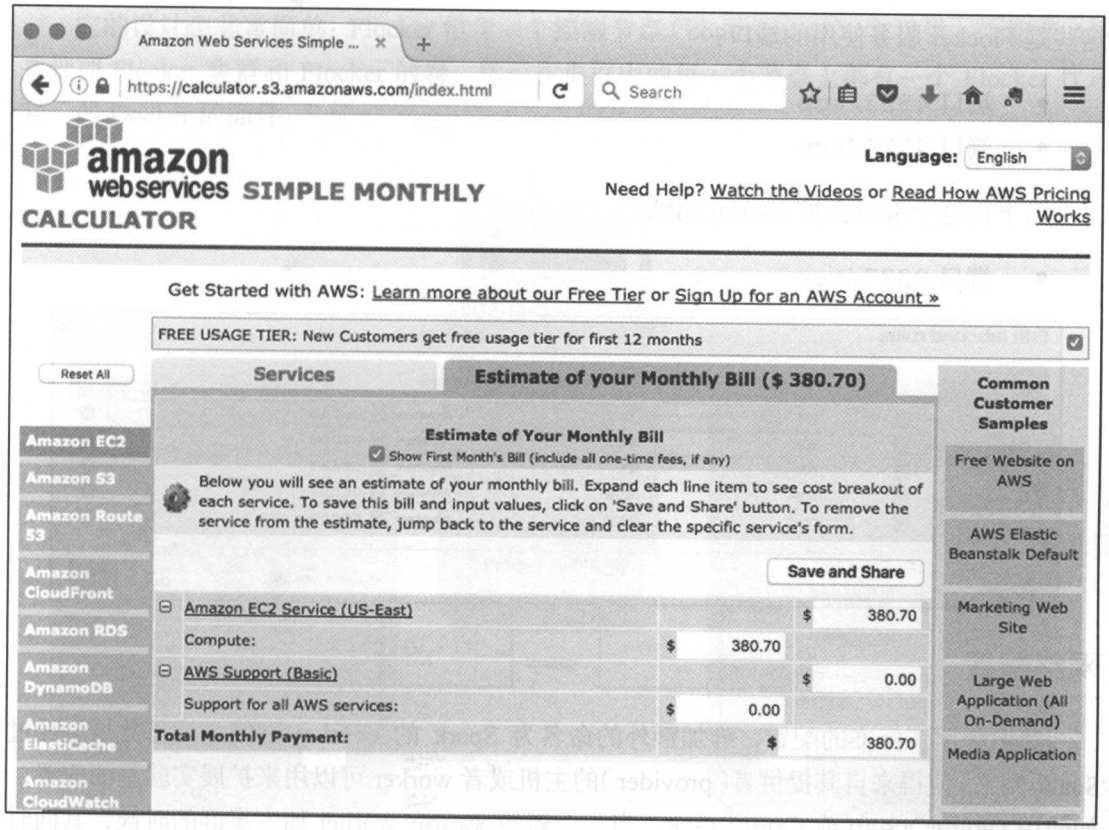
本书选择的操作系统版本是 Ubuntu 14.04 Trusty（指定了`--amazonec2-ami` 参数）。

安全组是标准的 `docker-machine`（我们会很快再次总结这方面的要求）。

选用 Flocker 1.15 的版本。

精确的 AMI ID（亚马逊系统映像 ID）可以在 https://cloud-images.ubuntu.com/locator/ec2/上检索到。

用 AWS 的计算器计算出的这个配置每个月的开销大概是 380 美元，不包含存储的使用费用。



让我们开始创建基础设施并把它们运行起来：

```
for i in `seq 101 110`; do
docker-machine create -d amazonec2 \
--amazonec2-ami ami-c9580bde \
--amazonec2-ssh-keypath keys/flocker \
--amazonec2-instance-type "t2.medium" \
aws-$i;
done
```

很快，基础设施应该就都启动并运行起来了。

安全组

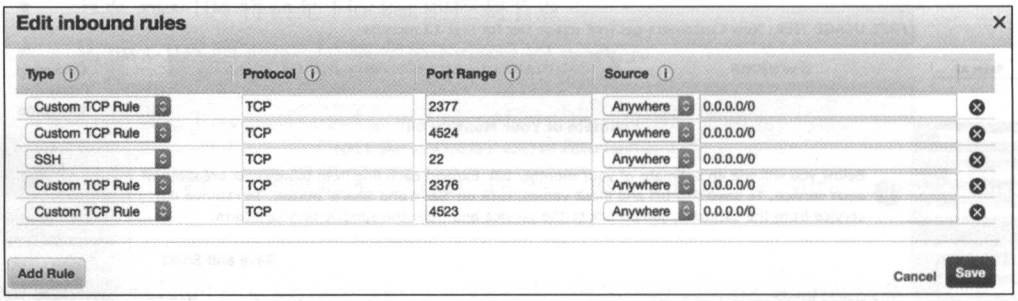
我们需要通过 EC2 的界面在这个项目（docker-machine）的安全组里面打开三个端口。

这些是 Flocker 服务使用的端口：

- 端口 4523/tcp
- 端口 4524/tcp

而下面这个端口会被 Swarm 用到：

- 端口 2377/tcp



网络配置

我们使用了标准的配置，外加额外的命名为 Spark 的 overlay 网络。流量数据会经过 Spark 网络，使得来自其提供者（provider）的主机或者 worker 可以用来扩展实验室的配置，如使用 DigitalOcean 或 OpenStack。当一个新的 Swarm worker 加入集群的时候，其同时会加入这个网络，然后被 Swarm 的服务用到。

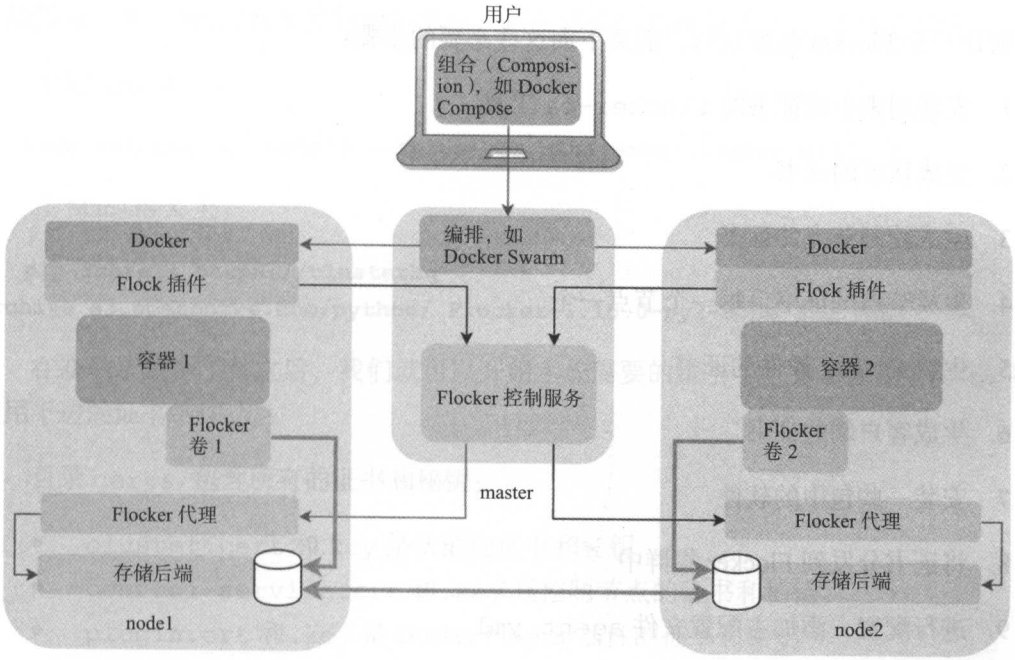
存储配置和架构

之前提到过，我们选择使用 Flocker（https://clusterhq.com/flocker/introduction/），它是一个顶级的 Docker 存储项目。ClusterHQ 是这么描述它的：

> Flocker 是一个开源的可以用作 Docker 化应用的数据卷管理器。Flocker 通过提供数据迁移的工具，能让运维团队运行容器化的有状态服务，如生产环境的 MySQL。与 Docker 的数据卷和单个服务器绑定的做法不同，一个 Flocker 的数据卷，又叫作 dataset，具有移植性并可以使用在集群中的任何一个容器上。

Flocker 支持的存储选项非常广泛，如 AWS EBS 到 EMC、NetApp、Dell、Huawei 的解决方案，它也支持 OpenStack Cinder 和 Ceph。

它的设计也非常简单：Flocker 包含一个控制节点（control node），它提供一个服务 API 来管理 Flocker 集群和 Flocker 的卷，然后在集群中的每一个节点上都有一个 Flocker 代理和一些 Docker 的插件。



要通过命令行使用 Flocker，你需要像下面这样运行 Docker，让它可以对 Flocker 的 myvolume 卷上的有状态数据进行读写操作，其容器内挂载路径为/data。

```
docker run -v myvolume:/data --volume-driver flocker image command
```

同时，你也可以通过使用 docker volume 命令来管理卷：

```
docker volume ls
docker volume create -d flocker
```

在本书这个架构中，我们将会把控制节点安装在 aws-104 上，这是专用的节点，然后在所有的其他节点上安装 Flocker 的代理（不包含节点 node-104）。

同时，我们也会安装 Flocker 的客户端，可以和 Flocker 的控制节点的 API 进行交互来

管理集群的状态和卷。为了方便，我们将会把它安装在 aws-104 上使用。

安装 Flocker

要让一个 Flocker 集群运行，需要下面这些必要的步骤：

1. 安装用来生成证书的 `flocker-ca` 工具
2. 生成认证的证书
3. 生成控制节点的证书
4. 生成节点的证书，每一个节点一个
5. 生成 Flocker 插件的证书
6. 生成客户端的证书
7. 安装一些包中的软件
8. 将证书分发到 Flocker 集群中
9. 进行配置，添加主配置文件 `agent.yml`
10. 配置主机上的数据包过滤器
11. 启动和重启动相关系统服务

对于小规模的集群，你可以手动执行这些步骤，但是这些都是重复和繁杂的劳动，所以我们将会使用一些简单的 Ansible playbook 来做这件事情，这些 playbook 已经放到 Github 的仓库中：https://github.com/fsoppelsa/ansible-flocker。

这些 playbook 都比琐碎，很可能还不足够生产环境的使用。你也能找到来自官方 ClusterHQ 的 Flocker role（可以查看 https://github.com/ClusterHQ/ansible-role-flocker），但是为了我们可以流畅地解释下去，我们还是选择使用第一个仓库中的 playbook，让我们先把它复制下来：

```
git clone git@github.com:fsoppelsa/ansible-flocker.git
```

生成 Flocker 证书

要生成证书必须要使用 `flocker-ca` 工具。安装说明可以查看 https://docs.clusterhq.com/en/latest/flocker-standalone/install-client.html。对于 Linux 发行版本来说，只需要安装一个软件包。对于 Mac OS X 发行版本来说，则需要通过 Python 的 `pip` 工具来安装。

在 Ubuntu 上：

```
sudo apt-get -y install --force-yes clusterhq-flocker-cli
```

在 Mac OS X 上：

```
pip install https://clusterhq-
archive.s3.amazonaws.com/python/ Flocker-1.15.0-py2-none-any.whl
```

在获取了这个工具之后，我们就可以开始生成需要的证书了。为了让任务简化，我们使用下边的证书结构：

目录 `certs/`包含所有的证书和秘钥：

- `cluster.cert` 和`.key` 是认证的证书和秘钥
- `control-service.crt` 和`.key` 是控制节点的证书和秘钥
- `plugin.crt` 和`.key` 是 Docker Flocker 插件的证书和秘钥
- `client.crt` 和`.key` 是 Docker 客户端的证书和秘钥
- `node-aws-101.crt` 和 `node-aws-101.key` 一直到 `node-aws-110.crt` 和 `node-aws-110.crt.key` 是节点的证书和秘钥，一个对应一个节点

下面是证书生成步骤：

1. 生成认证的证书：`flocker-ca initialize cluster`

2. 有了认证的证书和秘钥后，在相同目录下生成控制节点的证书：`flocker-ca create-control-certificate aws-101`

3. 然后生成插件的证书：`flocker-ca create-api-certificate plugin`

4. 然后生成客户端的证书：`flocker-ca create-api-certificate client`

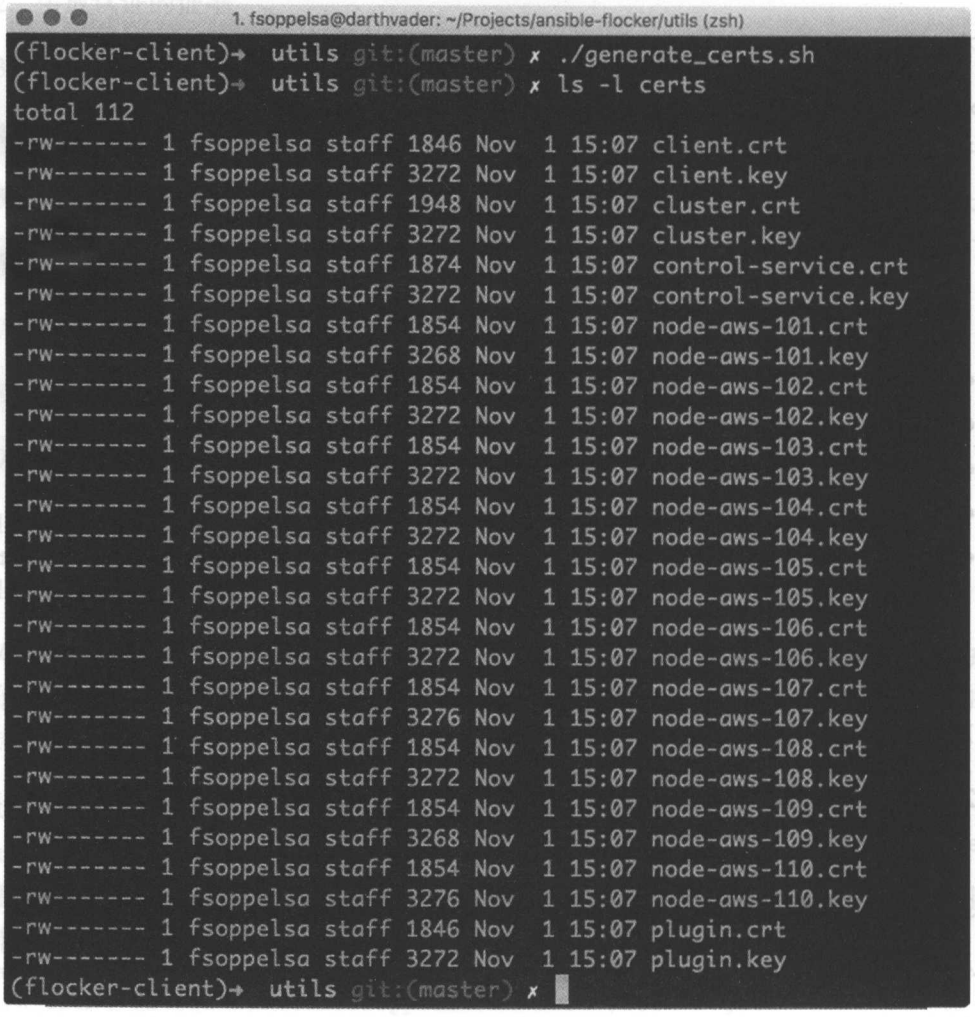
5. 最后为每一个节点生成证书：`flocker-ca create-node-certificate`

node-aws-X

当然，之前提过我们使用了偷懒的做法，我们将会使用 ansible-flocker 仓库中的 utility/generate_certs.sh 脚本来完成这些事情：

```
cd utils
./generate_certs.sh
```

在执行完这个脚本之后，/certs 目录就包含了所有的证书了：



安装软件

在每一个 Flocker 节点上，我们必须执行下面的这些步骤：

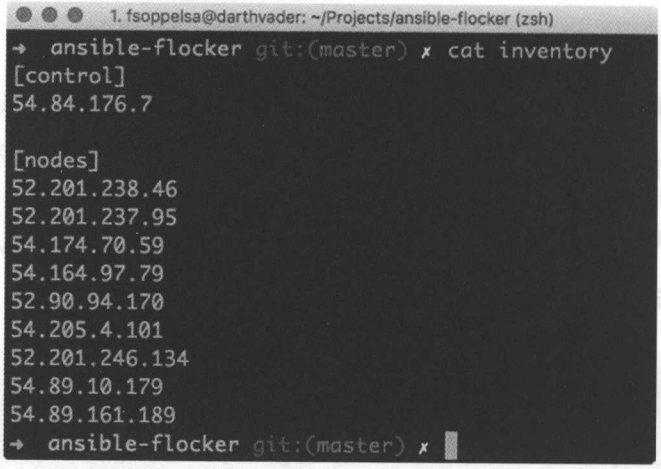
1. 把 ClusterHQ Ubuntu 仓库添加到 APT 的源列表
2. 更新包缓存（更新源索引）
3. 安装下面这些软件包

- `clusterhq-python-flocker`
- `clusterhq-flocker-node`
- `clusterhq-flocker-docker-plugin`

4. 创建一个目录`/etc/flocker`
5. 把 Flocker 的配置文件 `agent.yml` 复制到`/etc/flocker` 目录下
6. 把适合该节点的证书复制到`/etc/flocker` 目录下
7. 进行安全配置，启用 ufw，然后打开这些 TCP 端口：`2376`，`2377`，`4523`，`4524`
8. 启动系统服务
9. 重启 Docker 守护进程

同样的，我们希望计算机来帮我们完成这些事情。再次请 Ansible 出场设置好这些，我们去喝杯咖啡吧。

但是，首先我们必须指定好谁来作为 Flocker 的控制节点，谁来作为普通的节点。为此我们需要在 Ansible 的 `inventory` 文件中写好节点的 IP。`inventory` 是一个`.ini` 格式的文件，节点列表是它的必需内容。



然后，我们创建一个可以用来读取文件、证书和配置的 Ansible 目录，稍后我们将把其中的内容复制到节点：

```
mkdir files/
```

现在让我们把所有之前创建的证书，从 certs/目录复制到 files/目录下：

```
cp certs/* files/
```

最后，我们在 files/agent.yml 文件填充以下内容并作为 Flocker 的配置文件，你需要让配置文件适配 AWS 的区域并且相应地修改 hostname、access_key_id 和 secret_access_key：

```
control-service:
    hostname: "<Control node IP>"
    port: 4524
dataset:
    backend: "aws"
    region: "us-east-1"
    zone: "us-east-1a"
    access_key_id: "<AWS-KEY>"
    secret_access_key: "<AWS-ACCESS-KEY>"
version: 1
```

这是一个 Flocker 核心的配置文件，它会位于每一个节点的/etc/flocker 目录的下

边。你需要指定选用什么后端，并且配置好认证信息。在我们这个例子中，我们选用了基本的 AWS 选项——EBS，所以我们需要包含我们的 AWS 认证信息。

在文件 inventory、配置文件 `angent.yml` 和所有的证书都放置在 `files/` 目录下之后，我们可以继续下面的步骤。

安装控制节点

用来安装控制节点的 playbook 叫作 `flocker_control_install.yml`，这个 Ansible play 会执行一个安装脚本，并复制集群的证书、控制节点的证书和秘钥、节点的证书和秘钥、插件的证书和秘钥，并配置好 SSH、Docker 和 Flocker 需要的防火墙并打开相关端口，然后启动如下的系统服务：

- `flocker-control`
- `flockcer-dataset-agent`
- `flocker-container-agent`
- `flocker-docker-plugin`

最后，它会重启并刷新 `docker` 服务。

让我们来运行一下这个 playbook:

```
$ export ANSIBLE_HOST_KEY_CHECKING=False
$ ansible-playbook \
-i inventory \
--private-key keys/flocker \
playbooks/flocker_control_install.yml
```

安装集群节点

其他节点可以用类似的方法，用另外一个 playbook 安装：

```
$ ansible-playbook \
-i inventory \
--private-key keys/flocker \
playbooks/flocker_nodes_install.yml
```

这些步骤跟之前的大同小异，只是这个 playbook 不会复制证书，也不会启动

flocker-control 服务。在这些节点上只运行了 Flocker 的代理和 Flocker Docker 插件。让我们等待 Ansbile 执行完毕。

测试一切是否正常

要验证 Flocker 是否正确安装，我们可以登录到控制节点并检查 Flocker plugin 是否正在运行（可以通过检查它的.sock 套接字文件），然后我们可以通过如下命令安装 flockerctl 工具（可以查看 https://docs.clusterhq.com/en/latest/flocker-features/flockerctl.html 文档）：

```
$ docker-machine ssh aws-104
$ sudo su -
# ls /var/run/docker/plugins/flocker/
flocker.sock  flocker.sock.lock
# curl -sSL https://get.flocker.io |sh
```

我们需要设置一些 flockerctl 要使用的环境变量：

```
export FLOCKER_USER=client
export FLOCKER_CONTROL_SERVICE=54.84.176.7
export FLOCKER_CERTS_PATH=/etc/flocker
```

我们可以用下面的命令列出节点的卷（当然，现在还没有卷）：

```
flockerctl status
flockerctl list
```

```
2. root@aws-104: ~ (docker-machine)
root@aws-104:~# export FLOCKER_USER=client
root@aws-104:~# export FLOCKER_CONTROL_SERVICE=54.84.176.7
root@aws-104:~# export FLOCKER_CERTS_PATH=/etc/flocker
root@aws-104:~# flockerctl status
SERVER     ADDRESS
7df66dfd   172.31.5.7
409c3deb   172.31.2.37
885e9b3a   172.31.3.98
27bdd502   172.31.8.178
59430473   172.31.1.35
c12bfac9   172.31.6.150
e651fc75   172.31.8.52
d8759beb   172.31.0.213
05d393e4   172.31.8.38
63e67423   172.31.0.30

root@aws-104:~# flockerctl list
DATASET   SIZE   METADATA   STATUS   SERVER

root@aws-104:~#
```

现在，我们可以登录到集群的另外一个节点来检查 Flocker 集群的连通性（特别是插件和 agent 是否能够连接到控制节点并完成认证）。我们随机挑选了 `aws-108` 节点，创建一个卷并往里写一些数据：

```
$ docker-machine ssh aws-108
$ sudo su -
# docker run -v test:/data --volume-driver flocker \
busybox sh -c "echo example > /data/test.txt"
# docker run -v test:/data --volume-driver flocker \
busybox sh -c "cat /data/test.txt"
example
```

```
3. root@aws-108: ~ (docker-machine)
root@aws-108:~# docker volume ls
DRIVER              VOLUME NAME
root@aws-108:~# docker run -v test:/data --volume-driver flocker busybox sh -c "echo example > /data/test.txt"
root@aws-108:~# docker run -v test:/data --volume-driver flocker busybox sh -c "cat /data/test.txt"
example
root@aws-108:~#
```

如果我们回到控制节点，即 `aws-104`，我们可以通过 Docker 和 `flockerctl` 的相关列表命令来验证这个卷是否存在以及相应的持久化数据是否已经创建好：

```
2. root@aws-104: ~ (docker-machine)
root@aws-104:~# docker volume ls
DRIVER              VOLUME NAME
flocker             test
root@aws-104:~# flockerctl list
DATASET                              SIZE     METADATA    STATUS          SERVER
8577ed21-25a0-4c68-bafa-640f664e774e  75.00G   name=test   attached ✅   7df66dfd (172.31.5.7)

root@aws-104:~#
```

已经存在，好极了！那么我们现在可以删除已退出的测试容器，并在 Flocker 中删除用于测试的卷。现在我们已经准备就绪，可以开始安装 Swarm 了：

```
# docker rm -v ba7884944577
# docker rm -v 7293a156e199
# flockerctl destroy -d 8577ed21-25a0-4c68-bafa-640f664e774e
```

安装并配置 Swarm

现在我们就可以用最喜欢的方式来安装 Swarm，我们在前面的章节展示过这些方法。我们将选择 aws-101 和 aws-103 作为管理器，然后将剩下的除 aws-104 外的节点作为 worker。这个集群可以稍后进行扩展。根据实际，我们先保持它 10 个节点的规模。

```
1. fsoppelsa@darthvader: ~ (zsh)
→  ~ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
1ccgl8a2ftppqkktdycu4x4ml    aws-108   Ready   Active
1t9rg136bpjrcqzgnf9z2fhmj    aws-107   Ready   Active
29cii9ybw5qe6id7mwbxux2u6    aws-110   Ready   Active
2jkccfprthftnluvzclgjz3n6    aws-103   Ready   Active        Reachable
36pskhgjcmhlh9f18sgqyf2zi    aws-105   Ready   Active
4i9nuzux8xm3ao03u3odz5jec    aws-106   Ready   Active
4s333a5w59a8jotqlpf0pd5jn    aws-102   Ready   Active        Reachable
agsj97vrd11vhu6xvlu2mfob6    aws-109   Ready   Active
d46qo7hnyo7lubebe5s0jm5z5 *  aws-101   Ready   Active        Leader
→  ~
```

让我们添加一个专用的 overlay VxLAN 网络，命名为 `spark`：

```
docker network create --driver overlay --subnet 10.0.0.0/24 spark
```

为 Spark 添加一个卷

我们现在可以连接到任何一个 Docker 主机上，然后创建一个 75GB 大小的卷，用来保存一些持久化的 Spark 数据：

```
docker volume create -d flocker -o size=75G -o profile=bronze --
name=spark
```

这里需要讨论的一个选项是 `profile`。这是一个类似存储配置的东西（绝大多数情况是指速度）。如这个链接 https://docs.clusterhq.com/en/latest/flocker-features/aws-configuration.html#aws-dataset-backend 里面解释的，ClusterHQ 为 AWS EBS 维护了三个供 AWS 使用的 profile：

- Gold：EBS 配置的 IOPS / API，名为 io1。针对其容量配置的最大 IOPS，30 IOPS/GB，最大值可达到 20000 IOPS。
- Silver：EBS 用于普通目的的 SSD / API，名为 gp2。
- Bronze：EBS Magnetic / API，名为 standard

我们可以在 Flocker 的控制节点上用 `flockerctl list` 来检查这个卷是否已经添加了。

再次部署 Spark

让我们选择一个主机用来运行 Spark 单机的 manager，我们就选 `aws-105`，并给它添加好标签：

```
docker node update --label-add type=sparkmaster aws-105
```

其他节点用来安放我们的 Spark worker。

让我们在 `aws-105` 上启动 Spark master：

```
$ docker service create \
--container-label spark-master \
--network spark \
--constraint 'node.labels.type == sparkmaster' \
--publish 8080:8080 \
```

```
--publish 7077:7077 \
--publish 6066:6066 \
--name spark-master \
--replicas 1 \
--env SPARK_MASTER_IP=0.0.0.0 \
--mount type=volume,target=/data,source=spark,volume-driver=flocker
\
fsoppelsa/spark-master
```

首先，说一下上面用到的镜像。因为笔者发现 Google 的镜像有一些不太方便的地方（如去掉了一些环境变量，从而无法通过--env 参数来进行外部配置），所以笔者自己创建了 Spark 1.6.2 版本的 master 和 worker 镜像。

然后，说一下--network。我们通过这个选项告诉容器连接到用户命名为 Spark 的 overlay 网络。

最后，说一下存储：--mount，它支持 Docker 卷。我们指定为：

- 和卷一起使用
- 在容器内将卷挂载到 /data：target=/data
- 使用我们之前创建好的卷：source=spark
- 使用 Flocker 作为存储驱动

当你创建一个服务然后挂载某个卷的时候，如果卷不存在，那它会被创建。

当前的 Flocker 发布的版本只支持一个副本。原因是无法跨多个节点连接到 iSCSI/block 层的挂载点。所以在一个指定的时间点，只有一个服务可以使用副本数为 1 的卷。这让 Flocker 更加适用于保存和移动数据（这正是很多人用它的目的）。但是在这里我们使用它来展示一个小小的案例，其中持久化数据保存在 Spark master 容器中的/data 路径下。

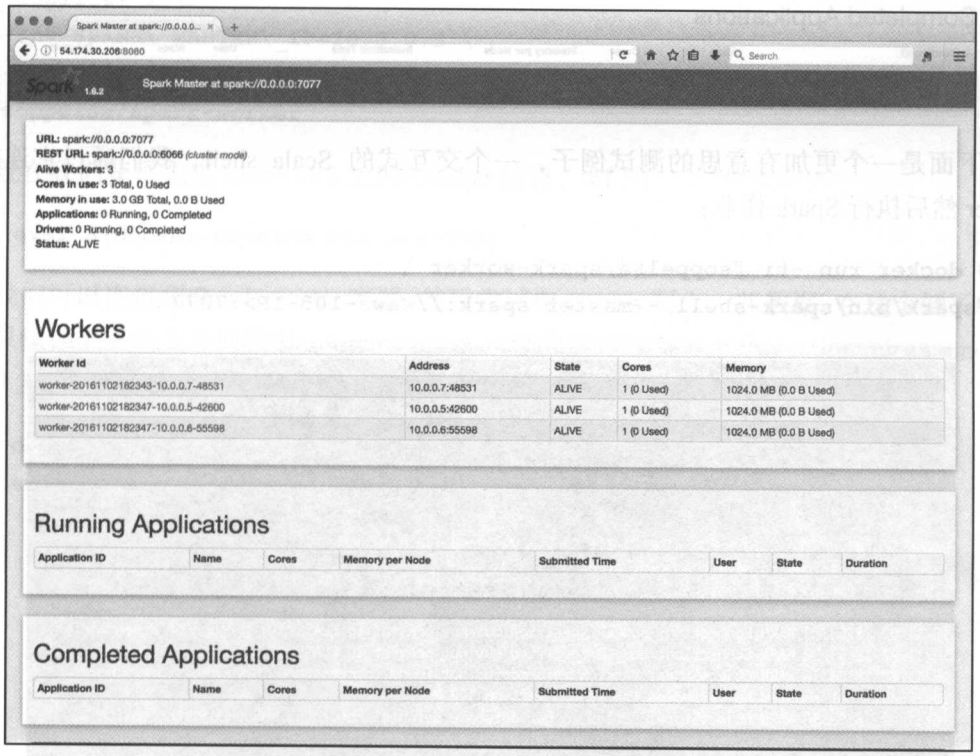
在上面的 Spark master 配置好之后，让我们来增添一些“马力”，即三个 Spark worker：

```
$ docker service create \
--constraint 'node.labels.type != sparkmaster' \
```

```
--network spark \
--name spark-worker \
--replicas 3 \
--env SPARK\_MASTER\_IP=10.0.0.3 \
--env SPARK\_WORKER\_CORES=1 \
--env SPARK\_WORKER\_MEMORY=1g \
fsoppelsa/spark-worker
```

我们在这里向容器传递了一些环境变量，来对资源的使用进行限制，让每个容器最多使用 1 核和 1GB 内存。

在几分钟之后，系统就会启动起来，我们通过访问 `aws-105` 的端口 `8080` 就会看到下面这个页面：



测试 Spark

让我们来访问 Spark shell 并运行一个 Spark 任务来检查一切是否已经启动并正常运行。

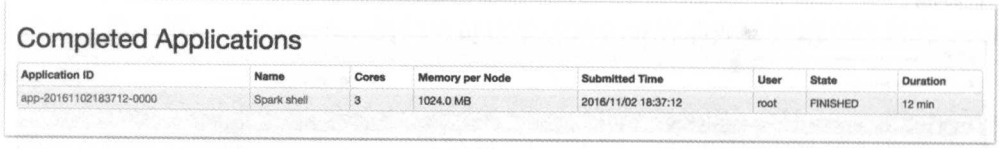
我们准备了一个容器，其中包含一些 Spark 工具，如 `fsoppelsa/spark-worker`，然后运行它，通过使用 Spark 的执行文件 `run-example` 来计算 Pi 的值：

```
docker run -ti fsoppelsa/spark-worker /spark/bin/run-example
SparkPi
```

在输出了一大堆信息之后，Spark 终于完成计算并返回下面的结果：

```
...
Pi is roughly 3.14916
...
```

如果，我们回到 Spark UI，可以看到我们的 Pi 应用已经成功地完成了。

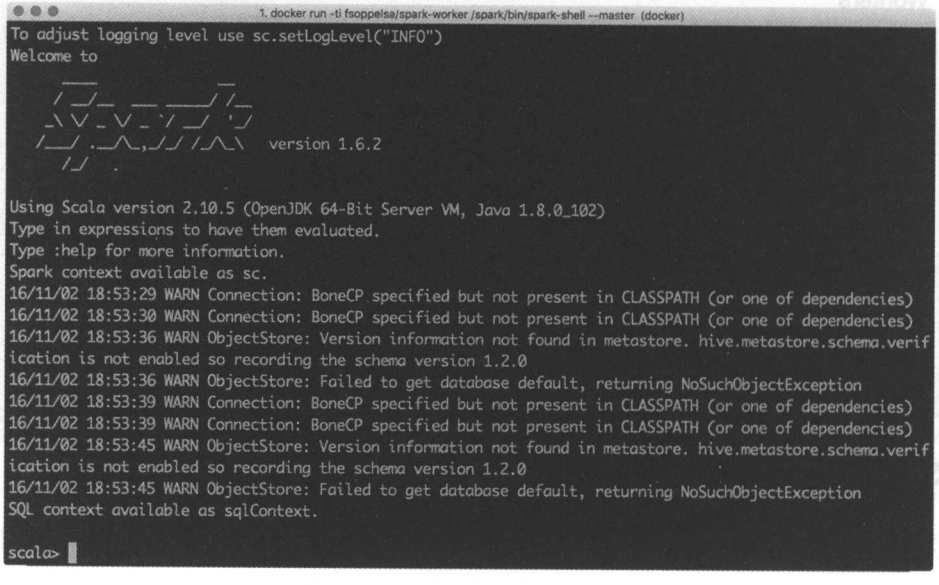


Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20161102183712-0000	Spark shell	3	1024.0 MB	2016/11/02 18:37:12	root	FINISHED	12 min

下面是一个更加有意思的测试例子，一个交互式的 Scala shell，我们通过它连接到 master 然后执行 Spark 作业：

```
$ docker run -ti fsoppelsa/spark-worker \
/spark/bin/spark-shell --master spark://<aws-105-IP>:7077
```



使用 Flocker 存储

为了完成本章教程的目标，我们现在运行一个例子，使用我们之前创建的卷，然后向 Spark 读写一些数据。

因为 Flocker 在复制因子上的限制，我们现在删除目前的由三个 worker 组成的集合，然后创建一个只包含一个 worker 的集合，并挂载 spark 卷：

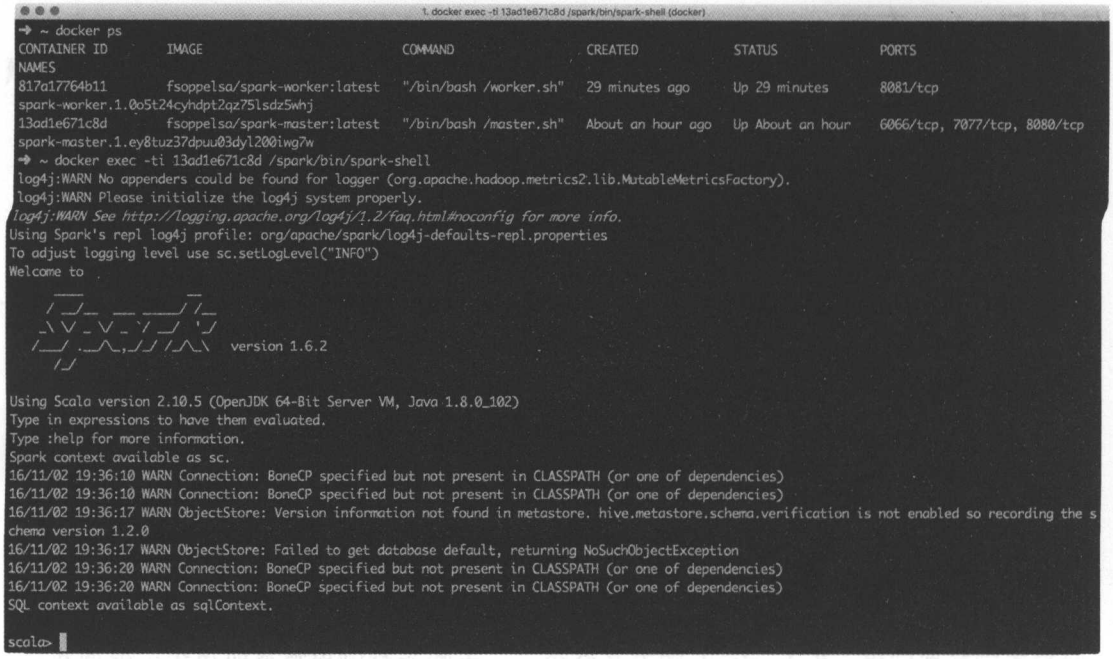
```
$ docker service rm spark-worker
$ docker service create \
--constraint 'node.labels.type == sparkmaster' \
--network spark \
--name spark-worker \
--replicas 1 \
--env SPARK\_MASTER\_IP=10.0.0.3 \
--mount type=volume,target=/data,source=spark,volume-driver=flocker\
fsoppelsa/spark-worker
```

现在我们获得主机 `aws-105` 上的 Docker 信息，如下：

```
$ eval $(docker-machine env aws-105)
```

我们可以连接到 Spark master 容器，然后尝试向 `/data` 写入一些数据。在这个例子中，我们只保存了一些文本型数据（lorem ipsum 示例内容可以在 http://www.loremipsum.net/获取到）到 `/data/file.txt`。

```
$ docker exec -ti 13ad1e671c8d bash
# echo "the content of lorem ipsum" > /data/file.txt
```



然后，我们连接到 Spark shell，并执行一个简单 Spark 作业：

1. 加载 `file.txt`。
2. 将它包含的单词映射成它们的出现次数。
3. 将结果保存到`/data/output`。

```
$ docker exec -ti 13ad1e671c8d /spark/bin/spark-shell
...
scala> val inFile = sc.textFile("file:/data/file.txt")
scala> val counts = inFile.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey(_ + _)
scala> counts.saveAsTextFile("file:/data/output")
scala> ^D
```

```
scala> val inFile = sc.textFile("file:/data/file.txt")
inFile: org.apache.spark.rdd.RDD[String] = file:/data/file.txt MapPartitionsRDD[1] at textFile at <console>:27

scala> val counts = inFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:29

scala> counts.saveAsTextFile("file:/data/output")

scala> Stopping spark context.
```

现在，让我们在任何一个 Spark 节点上启动一个 `busybox` 容器，然后检查 `spark` 卷，确认其中是否包含输出的内容。下面是我们运行的代码：

```
$ docker run -v spark:/data -ti busybox sh
# ls /data
# ls /data/output/
# cat /data/output/part-00000
```

```
➡ ~ docker run -v spark:/data -ti busybox sh
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
56bec22e3559: Pull complete
Digest: sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
Status: Downloaded newer image for busybox:latest
/ # ls /data
file.txt  output
/ # ls /data/output/
_SUCCESS    part-00000  part-00001
/ # cat /data/output/part-00000
(vulputate,1)
(interdum,1)
(ullamcorper,,1)
(porta,1)
(ac,3)
(neque,1)
(lacus.,1)
(ante,1)
```

上图显示了其输出符合预期。关于 Flocker 卷有意思的一点是它们甚至可以从一个主机移动到另外的主机，另外还可以执行很多可靠的操作。如果你正在选择一个优秀的 Docker 存储解决方案，那 Flocker 不失为一个好的选择。例如 Swisscom Developer cloud（http://developer.swisscom.com/）在生产环境中使用 Flocker，让用户可以用 Flocker 的技术作为 MongoDB 等数据库的存储后端。之后的 Flocker 发布版本将会集中在代码库简化上，

使之变得更加精益和可靠，如内置高可用，快照、证书分发和能在容器内简易部署的代理都是接下来的关注要点。所以，未来值得期待。

伸缩 Spark

现在我们将介绍 Swarm 最了不得的特性，即 `scale` 命令。我们首先回到 Flocker 之前的配置，并销毁 `spark-worker` 服务，然后创建一个副本因子为 3 的服务：

```
aws-101$ docker service create \
--constraint 'node.labels.type != sparkmaster' \
--network spark \
--name spark-worker \
--replicas 3 \
--env SPARK_MASTER_IP=10.0.0.3 \
--env SPARK\_WORKER\_CORES=1 \
--env SPARK\_WORKER\_MEMORY=1g \
fsoppelsa/spark-worker
```

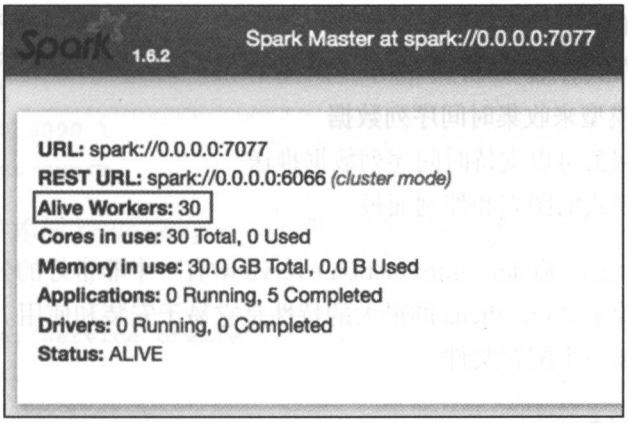
现在，我们对服务进行向上伸缩，使之有 30 个 Spark worker。使用的命令如下：

```
aws-101$ docker service scale spark-worker=30
```

几分钟过后，在镜像下载好后，我们可以再次检查服务的状态：

```
1. fsoppelsa@yoda: ~ (zsh)
➜ ~ docker service ls
ID            NAME          REPLICAS  IMAGE                    COMMAND
bive3l6t0s73  spark-master  1/1       fsoppelsa/spark-master
dg27bmgyzk5n  spark-worker  3/3       fsoppelsa/spark-worker
➜ ~ docker service scale spark-worker=30
spark-worker scaled to 30
➜ ~ docker service ls
ID            NAME          REPLICAS  IMAGE                    COMMAND
bive3l6t0s73  spark-master  1/1       fsoppelsa/spark-master
dg27bmgyzk5n  spark-worker  30/30     fsoppelsa/spark-worker
➜ ~
```

从 Spark UI 中我们可以看到：



Scale 命令可以用来向上伸缩或者向下伸缩副本。目前，还没有自动化的机制可以进行自动化伸缩，或者将负载分发到新创建的节点上。但是，我们可以使用自定义的脚本来实现这个功能。我们可以期待 Swarm 中很快会集成这个功能。

监控 Swarm 托管的应用

我（Fabrizio）在 2016 年 8 月关注过一个 Reddit 上的讨论（https://www.reddit.com/r/docker/comments/4zous1/monitoring_containers_under_112_swarm/），其中用户抱怨到新的 Swarm Mode 监控起来更加困难。

如果暂时没有官方的 Swarm 监控方案，一种最流行的通过组合新技术的方案是：用 Google 的 cAdvisor 来收集数据，用 Grafana 来展示图表，然后用 Prometheus 作为数据模型。

Prometheus

Prometheus 团队对这个产品的描述是：

> Prometheus 是一个开源的系统监控和报警工具集，起初其是在 SoundCloud 中创建的。

它的主要特性有：

- 多维数据模型

- 灵活的查询语言
- 不依赖分布式存储
- 通过 Pull 模型来收集时间序列数据
- 通过一个网关可以支持时间序列数据推送
- 支持多个模式的图表和控制面板

在 https://prometheus.io/docs/introduction/overview/有一个非常好的分享，我们在这里不打算重述。在我们看来 Prometheus 的最大的特性是它易于安装和使用。Prometheus 只包含一个二进制文件外加一个配置文件。

安装一个监控系统

可能事情很快会有所改观，所以我们在这里大概地描述一下为 Swarm 搭建一个监控系统的过程，这里的例子是在 Docker 1.12.3 版本中测试可用的。

首先，我们需要创建一个新的 overlay 网络，为了让它不和 `ingress` 和 `spark` 网络冲突，我们把它命名为 `monitoring`：

```
aws-101$ docker network create --driver overlay monitoring
```

然后，我们以 `global` 模式启动一个 cAdvisor 服务，这意味着 cAdvisor 容器会在每一个 Swarm 节点上运行。我们在容器里挂载了一些系统路径，从而使它们可以被 cAdvisor 访问到：

```
aws-101$ docker service create \
  --mode global \
  --name cadvisor \
  --network monitoring \
  --mount type=bind,src=/var/lib/docker/,dst=/var/lib/docker \
  --mount type=bind,src=/,dst=/rootfs \
  --mount type=bind,src=/var/run,dst=/var/run \
  --publish 8080 \
  google/cadvisor
```

接着，我们使用 `basi/prometheus-swarm` 镜像来配置好 Prometheus：

```
aws-101$ docker service create \
```

```
--name prometheus \
--network monitoring \
--replicas 1 \
--publish 9090:9090 \
prom/prometheus-swarm
```

之后，我们继续添加 node-exporter 服务（我们还是使用 global 模式，它必须运行在每一个节点上）：

```
aws-101$ docker service create \
  --mode global \
  --name node-exporter \
  --network monitoring \
  --publish 9100 \
  prom/node-exporter
```

最后，我们启动 Grafana，副本数为 1：

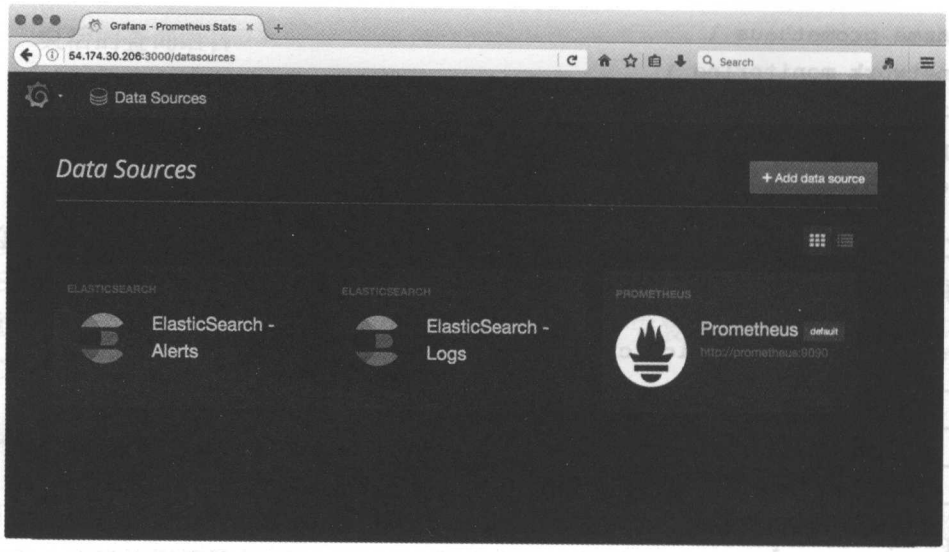
```
aws-101$ docker service create \
  --name grafana \
  --network monitoring \
  --publish 3000:3000 \
  --replicas 1 \
  -e "GF_SECURITY_ADMIN_PASSWORD=password" \
  -e "PROMETHEUS_ENDPOINT=http://prometheus:9090" \
  grafana/grafana
```

在 Grafana 中导入 Prometheus

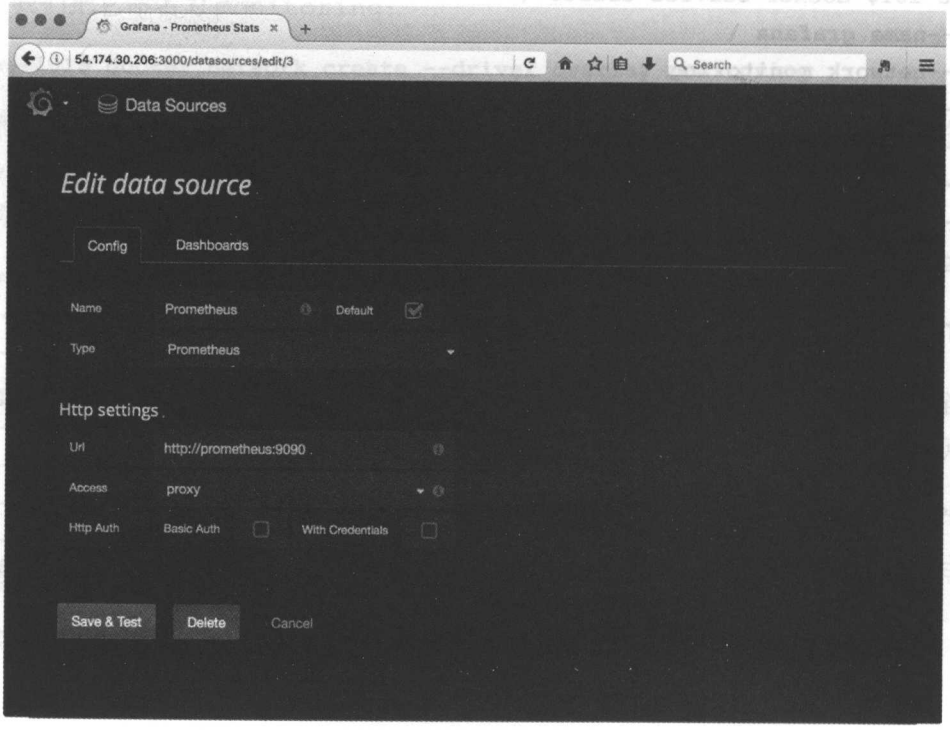
当 Grafana 正常运行的时候，我们可以通过使用下面的认证信息来登录到 Grafana 运行的节点，端口是 3000，来获取关于 Swarm 健康状况的漂亮图表：

```
"admin":"password"
```

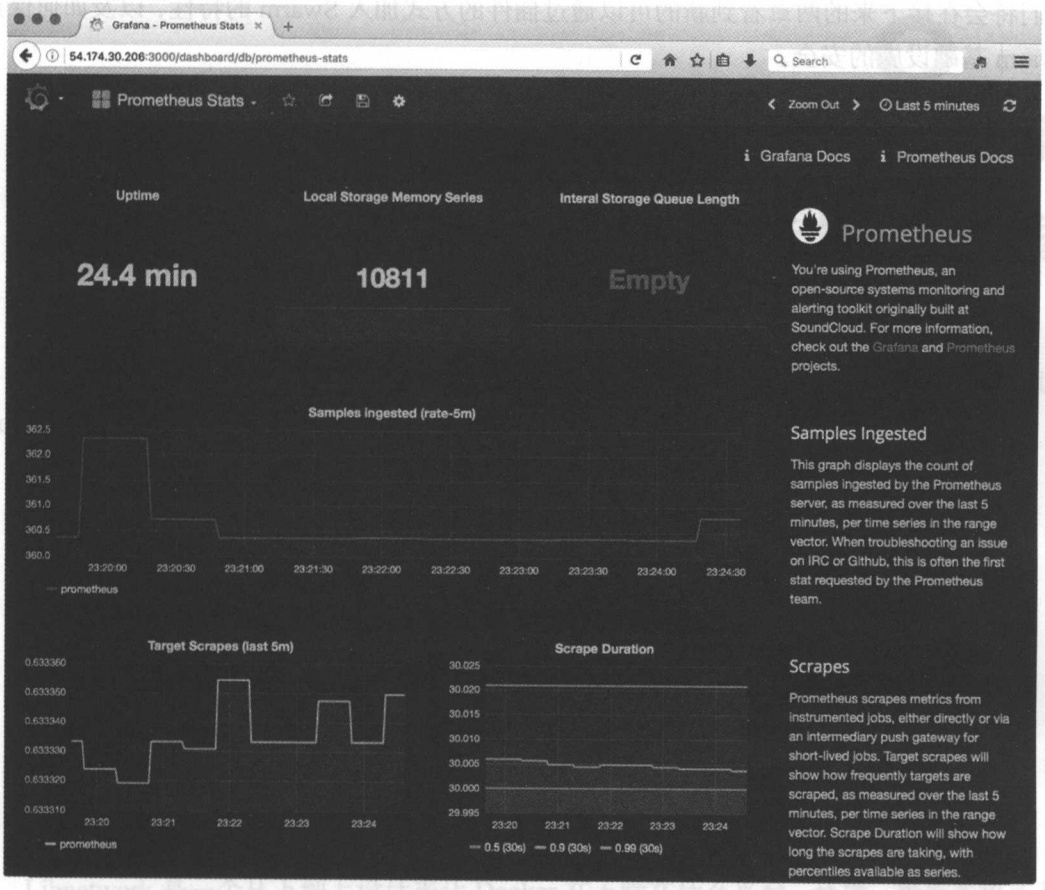
作为管理员，我们可以点击 Grafana 的 logo，然后来到 Data Source，并添加 Prometheus：



接着会出来一些选项，但是映射关系已经在了，所以直接单击“Save & Test”按钮并继续。



现在，我们可以回到控制面板然后单击 Prometheus，然后呈现在我们眼前的就是 Grafana 的主要面板信息：



在这里，我们再一次利用了开源社区的产品，然后用一些简单的命令，结合各种个人偏好的技术，就获得了想要的结果。目前 Docker Swarm 和它的应用的监控是一个完全开放的领域，我们期待其中会有了不起的创新出现。

本章小结

在本章中，我们通过 Flocker 向 Swarm 基础设施增加存储容量，然后设置了专门的网络来打造我们的示例应用，即一个 Spark 集群。我们随后对这个应用进行了使用，并且让

它可通过添加节点（同时针对新的提供者，如 DigitalOcean）来轻松地扩展。在使用了 Spark 和 Flocker 后，我们最后介绍了 Prometheus 和 Grafana，来对 Swarm 的健康状态进行监控。我们将会在接下来的两章看到新的可以通过插件的方式加入 Swarm 的特性，以及如何保证 Swarm 基础设施的安全。

第 8 章

Swarm 附加特性的探索

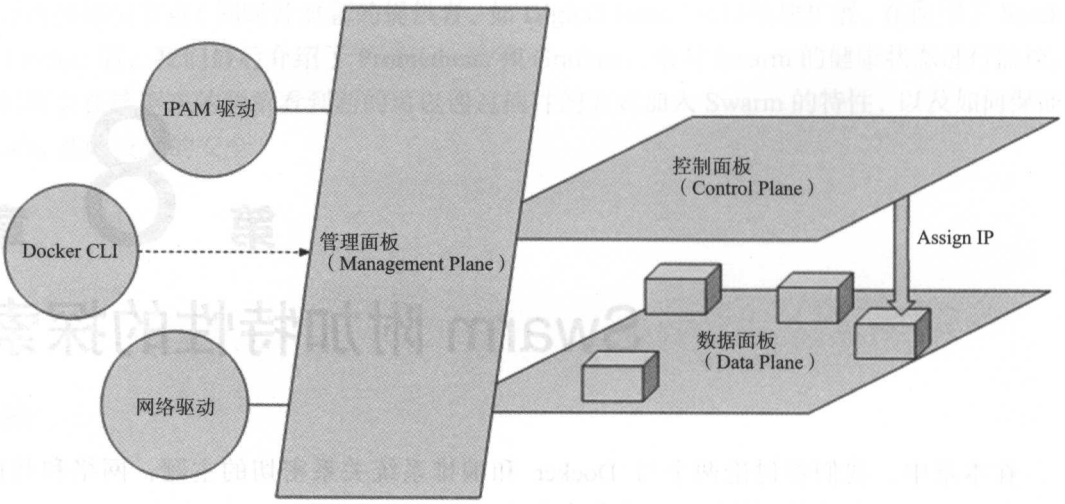
在本章中，我们将讨论两个与 Docker 和编排系统关系密切的主题：网络和共识（consensus）。我们会特别关注下面这些内容：

- Libnetwork 基础
- Libnetwork 安全基础
- 路由网格（Routing mesh）
- overlay 网络
- 网络控制面板（Network Control Plane）
- Libkv

Libnetwork

Libnetwork 是一个从下到上设计来让 Docker 在不管在什么平台、环境、操作系统或基础设施中都能正常运转的网络栈。Libnetwork 不仅是一个网络驱动的接口，也不仅是一个用来管理 VLAN 或者 VXLAN 的库，它还能做更多的事情。

Libnetwork 是一个完整的网络栈，包含三个面板：**管理面板**（Management Plane）、**控制面板**（Control Plane）和**数据面板**（Data Plane），如下图所示：



- **管理面板**可以让用户、运维人员或工具来管理网络的基础设施。这些操作包括了网络监控。管理面板代表了 Docker 网络的用户体验，并提供了 API。它也能通过管理插件来扩展，如 IPAM 插件允许用户控制如何向每一个容器分配 IP。
- **控制面板**的实现基于带有作用域的 gossip 协议，并直接加入了服务发现和加密秘钥分发的功能。
- **数据面板**简单地来说负责两个端点之间网络数据包的移动。网络插件作用于每一个数据面板。它有一些默认的内置驱动。例如，在之前章节我们见过的 overlay 驱动，它直接利用了 Linux 和 Windows 系统的内核特性，因此对于这种类型的网络来说，并没有驱动代码的实现。桥接、IPVLAN 和 MacVLAN 驱动也是如此。相反的，其他第三方网络就需要以插件的方式用自己的方式实现。

Libnetwork 遵循了 Docker 一贯的用户体验精神，这意味着它的组件应该能在任何环境中正常运作，同时网络栈也必须要有可移植性。而为了让 Docker 的网络栈具有可移植性，它必须具有可靠的设计和实现。例如，管理面板不能被其他任何组件控制，如果允许这么做，在应用的环境改变的时候网络栈可能就会崩溃。

Networking 插件

数据面板拥有可插拔设计。实际上，它只能被内置的或外部的插件管理。例如，MacVLAN 是以插件的方式实现然后添加进 Docker 1.12 版本中的，这让它不会对系统的其

他部分造成影响。

最重要的一点是，我们可以在同一个网络栈中有多个驱动和插件，它们工作的时候不会彼此冲突。所以，通常情况下在 Swarm 中，我们可以在集群中同时运行一个 Overlay 网络、一个桥接网络和一个 host driver。

容器网络模型

Libnetwork 满足了 Docker Swarm 运行 Docker 分布式应用的需求。Libnetwork 的基础是一个叫作**容器网络模型**（Container Networking Model，CNM）的模型。这个基础模型包含清晰的定义，描述了容器应该如何连接到指定的网络。CNM 包含三个组件：

- **沙盒（Sandbox）**：作为一种隔离，包含了容器网络栈的配置。
- **端点（Endpoint）**：这是一个连接点，只属于一个网络和一个沙盒。
- **网络（Network）**：这是一个端点组，它们之间允许自由通信。一个网络包含一个或者多个端点。

驱动是数据面板的代表。每一个驱动，不论是 overlay、bridge 或者 MacVLAN 都是以插件的形式存在的。每一个插件都运行在自己特定的数据面板里。

在系统中默认有一个内置的 IPAM。它有很重要的意义，正如在传统系统中的做法一样，我们需要容器有一个 IP 地址来和它进行联系。因为每一个容器都必须添加一个 IP 地址，所以有必要存在这样一个内置的 IPAM 系统。我们也要求能定义子网和 IP 地址范围。同时，系统的 IPAM 具有可插拔的设计，这意味着它允许我们使用自己的 DHCP 驱动，或者让系统连接到一个已经存在的 DHCP 服务器。

正如之前提到的一样，Libnetwork 开箱即有对多主机网络的支持。多主机网络中值得讨论的组件是它的数据面板和控制面板。

目前包含在 Docker 1.12 版本中的控制面板使用 gossip 机制来作为节点的通用发现系统。这个基于 gossip 协议的网络工作在另外一层，与 Raft 共识系统平行。基本说来，我们同时有两个不同的成员机制（membership mechanism）一起工作。Libnetwork 允许来自其他插件的驱动共同使用控制面板。

下面是 Libnetwork 控制面板的特性：

- 它开箱即是安全的并已加密
- 可以被每一个数据面板使用
- 它开箱即有原生的服务发现和负载均衡支持

Docker 1.12 版本在 Swarm 中实现了基于 VIP 的服务发现。这个服务通过将一个容器的虚拟 IP 地址映射到 DNS 记录，然后通过 gossip 协议共享所有的 DNS 记录。在 Docker 1.12 版本中，随着 service（服务）概念的引入，这直接与发现的概念很搭配。

而在 Docker 1.11 版本和之前的版本中，则需要使用容器的名字和别名来“模拟”服务发现，然后需要通过 DNS 轮询才能做到某些很原始的负载均衡。

Libnetwork 遵循了“自带动力但允许更换（battery included but removable）”的原则，这个例子就是插件系统。在未来，Libnetwork 将会逐渐地扩展插件系统，并覆盖网络的其他部分，如负载均衡。

加密和路由矩阵

正如之前提到的，在 Libnetwork 的核心使用了 CNM 模型。在 Swarm Mode 中，libnetwork 是处于一种能感知集群的模式（cluster-aware mode），并且无需外部的键值存储就能支持多主机网络。overlay 网络很自然地适合这种模型。同时引入了数据面板和控制面板的加密。有了加密的控制面板，VXLAN 上的路由信息，如哪些容器拥有哪些 MAC 地址和 IP 地址都自动地被加密了。同时，有了路由矩阵（Routing Mesh），CNM 提供了一种去中心化的机制，能让你从集群中的任何一个 IP 地址访问你的服务。当一个请求从集群外部到达集群中任何一个节点的时候，流量将会被路由到一个可用的容器。

MacVLAN

在 Docker 1.12 版本中新加入的驱动是 MacVLAN，它是一个高性能的驱动，它让 Docker 网络连接到已有的 VLAN，如企业中的 VLAN，让一切继续工作。我们遇到过一个场景就是我们需要逐渐地将工作负载从原来的 VLAN 迁移到 Docker 上，然后 MacVLAN 帮助我们将 Docker 集群连上原来的 VLAN。这会让 Docker 网络与底层的网络整合起来，然后让

容器继续在同一个 VLAN 中正常工作。

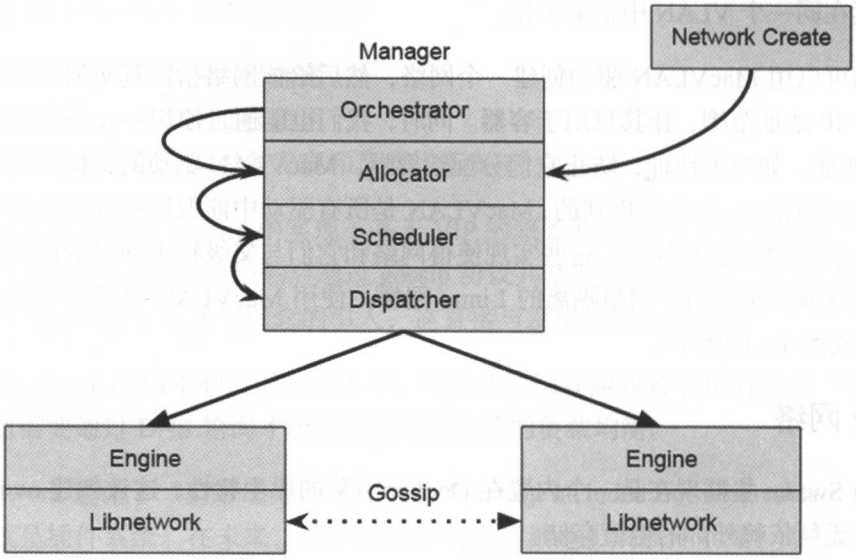
我们可以用 MacVLAN 驱动创建一个网络，然后给该网络指定真实的子网。我们也能指定一个 IP 地址范围，让其只用于容器。同时，我们也能通过使用`--aux-address`排除一些 IP 地址，如网关地址，防止它们分配给容器。MacVLAN 驱动的父接口是我们想要该网络连接到的接口。如之前提到的，MacVLAN 是所有驱动中能取得最好结果的一个驱动。它的Linux实现特别的轻量级，这些实现使得网络和它们与父级物理网络的连接分离开来，而不是实现成传统的用于网络隔离的 Linux 网桥。使用 MacVLAN 驱动需要 Linux 内核是 3.9~3.19 或者 4.x 的版本。

overlay 网络

因为 Swarm 集群现在是一个内置在 Docker 引擎的原生特性，这让创建 overlay 网络非常容易，无须依赖外部的键值存储。

管理器节点负责网络的状态管理。所有的网络状态都保存在 Raft 日志中。Swarm mode 中的 Raft 实现和外部键值存储的区别是，内置的 Raft 比外部实现有更高的性能。我们自己的实验显示使用外部键值存储能达到的规模大概在 100~250 个节点，而在 Swarm3k 的活动中，内置的 Raft 却帮助我们把系统伸缩到了 4700 个节点。这是因为外部的 Raft 存储基本上都有很高的网络延迟。当我们需要就某状态达成一致的时候，会因为网络往返通信遇到问题，而内置的 Raft 存储却只需要访问内存。

在过去，当我们想要执行任何与网络相关的动作时，如给容器分配 IP 地址，会出现很严重的网络延迟，因为我们总是在跟外部的存储通信。对于内置的 Raft 来说，当我们需要就某个值达成共识的时候，我们可以立即完成访问内存的存储（in-memory store）。



当我们用 overlay 驱动创建一个网络的时候，像下面这样：

```
$ docker network create --driver overlay --subnet 10.9.0.0/24 mh_net
```

该命令会和 allocator（分配器）进行通信。然后会进行子网保留（subnet reservation）。这个例子中该子网是 `10.9.0.0/24` ，然后在管理器主机内存中立即会就相关的值达成一致。如果稍后我们希望创建一个 service，那么我们会将该 service 连接到该网络。当我们像下面这样创建 service 的时候：

```
$ docker service create --network mh_net nginx
```

编排器会为该 service 创建很多的任务（容器）。然后每一个创建的任务都会指定一个 IP 地址。在这个 IP 指定的过程中，分配也会发生作用。

在任务创建完成后：

- 任务会获得一个 IP 地址。
- 它的网络相关信息将会被提交（commit）到 Raft 的日志存储中。
- 在 allocation 完成提交后，调度器会将任务移动到另外一个状态。
- Dispatcher 会将每一个任务分发到一个 worker 节点上。
- 最后，与该任务关联的容器会在 Docker 引擎上运行。

如果一个任务不能分配到它的网络资源，它会阻塞在已分配（allocated）状态，然后不会被调度。这是一个和之前 Docker 版本很重要的不同点，即在 Swarm Mode 的网络系统中，分配状态的概念是明确的。有了它，能大大改善系统整体的分配循环。当我们谈到分配的时候，我们不仅指的是 IP 地址的分配，我们也指相关的驱动产出（driver artifacts）的分配。对于一个 overlay 网络来说，它需要预留一个 VXLAN 的标识符，这是一组为每一个 VXLAN 准备的全局标识符。这个标识符预留的动作是由网络分配器（Network Allocator）完成的。

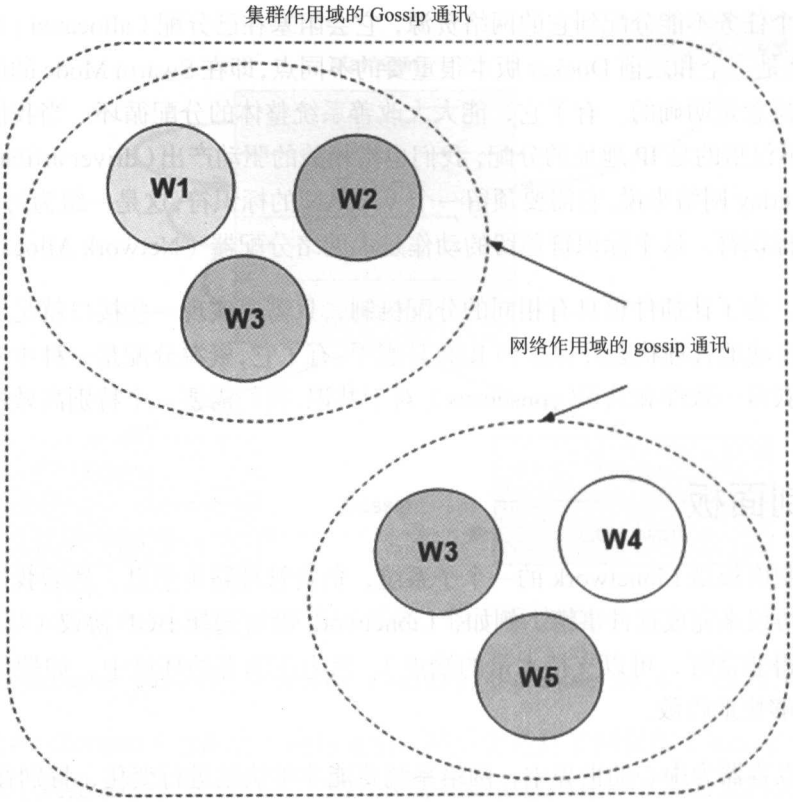
在将来，为了让插件也具有相同的分配机制，只需要实现一些接口就足够了，然后让 Libnetwork 自动地管理状态并保存到 Raft 日志中。有了它，资源分配是一种中心化的方式，因此我们能获得一致性和共识（consensus）。对于共识，我们需要一个特别高效的共识协议。

网络控制面板

网络控制面板是 Libnetwork 的一个子系统，负责管理路由信息，然后我们需要一个能快速收敛的协议来完成这件事情。例如，Libnetwork 没有选择 BGP 协议（尽管 BGP 在伸缩性方面做得非常好，可以支持大量的端点），因为在动态的环境中，如软件容器环境，BGP 无法足够快地收敛。

在一个以容器为中心的世界中，网络系统要能非常快地进行变化，特别在新的 Docker service 模型中，有大批量和快速的 IP 指定需求。我们想要路由信息快速地收敛，对于大规模场景尤其是这样的情况，例如当容器的规模超过 10000 的时候。在 Swarm2k 和 Swarm3K 的试验中，我们的确达到了一次运行 10000 个容器。特别是在 Swarm3k 中，我们在 Ingress 的负载均衡网络中启动了 4000 个 Nginx 容器。如果没有优良的实现，在这个数目的伸缩规模，集群将无法正常运作。

为了解决这个问题，Libnetwork 团队决定将 gossip 协议包含在网络控制面板中。这个协议的内部算法是这样的：它会选择三个邻居，然后将相同的信息扩散（propagate）出去；对于 Libnetwork 来说，这包括路由和其他网络相关的信息。gossip 协议会反复执行这个操作，直到每一个节点都共享了相同的信息。使用这种技术，整个集群将会很快地（几秒的时间）收到信息。



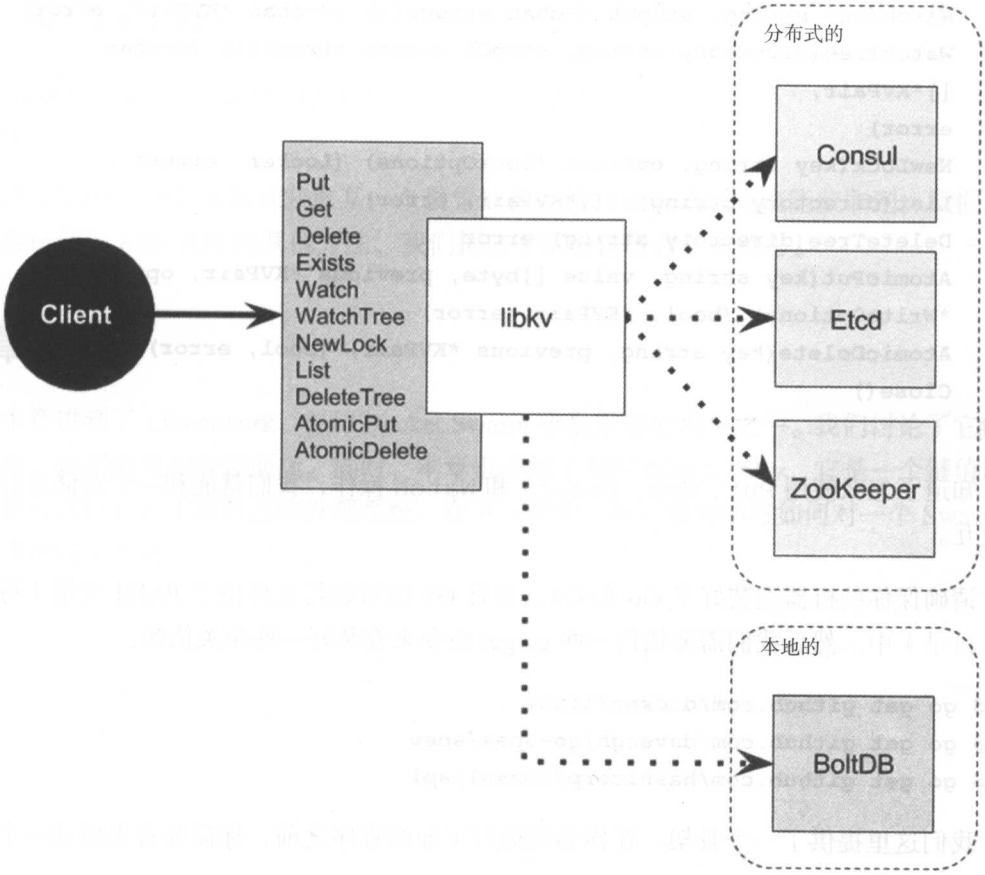
无论如何，整个集群并不是所有时间都需要共享相同的信息。不是集群中的每一个节点都需要知道所有网络的信息。只有位于某特定网络中的节点才需要知道自己所在的网络的信息。为了优化 Libnetwork 的这一点，团队实现了两种作用域：集群作用域的 gossip 通信和网络作用域的 gossip 通信。到目前为止，我们解释的都属于集群作用域的 gossip 通信，而网络作用域的 gossip 通信会将网络信息限制在某个特定的网络中。当一个网络进行扩展，并覆盖更多的节点的时候，它的 gossip 作用域的广播会同时覆盖到它们。

这个动作是建立在 Docker 的 CNM 之上的，因此依赖网络的抽象。从上图中，左边的网络包括节点 W1、W2 和 W3，然后右边的网路包括 W3、W4 和 W5。左边的网络在 gossip 执行的时候只有 W1、W2 和 W3 会知道它的路由信息。你可能已经观察到，W3 同时位于两个网络中，因此它同时会接收到来自左边和右边网络的路由信息。

Libkv

libkv 是一个统一的库，其可以和不同的键值存储后端交互。libkv 原本是 Docker Swarm v1 第一个开发版本的一部分。之后，所有的键值存储发现服务的代码都被重构并移动到了 www.github.com/docker/libkv 仓库。

libkv 允许你执行 CRUD 操作并监听不同后端的键值，所以针对所有高可用键值存储来说都可以使用同一份代码，不论是 Consul、Etcd 或者 Zookeeper，正如下图所示。在本书撰写的时候，libkv 也支持一个用 BoltDB 实现的本地存储。



如何使用 libkv

要开始使用 libkv，我们需要首先知道如何调用它的 API。这里是 libkv 的 Go 源码中的 Store 接口，每一个存储的实现都需要实现它：

```
type Store interface {
  Put(key string, value []byte, options *WriteOptions) error
  Get(key string) (*KVPair, error)
  Delete(key string) error
  Exists(key string) (bool, error)
  Watch(key string, stopCh <-chan struct{}) (<-chan *KVPair, error)
  WatchTree(directory string, stopCh <-chan struct{}) (<-chan
  []*KVPair,
  error)
  NewLock(key string, options *LockOptions) (Locker, error)
  List(directory string) ([]*KVPair, error)
  DeleteTree(directory string) error
  AtomicPut(key string, value []byte, previous *KVPair, options
  *WriteOptions) (bool, *KVPair, error)
  AtomicDelete(key string, previous *KVPair) (bool, error)
  Close()
}
```

知道了怎么使用 Put、Get、Delete 和 Watch 操作，我们就能和一个存储进行基本的交互。

请确保你的机器安装好了 Go 和 Git，并且 Git 的可执行文件位于 PATH 变量（可执行文件目录）中。然后我们需要执行一些 go get 命令来安装好一些相关依赖：

```
$ go get github.com/docker/libkv
$ go get github.com/davecgh/go-spew/spew
$ go get github.com/hashicorp/consul/api
```

我们这里提供了一个骨架。在你尝试运行下面的程序之前，你需要首先启动一个单节点的 Consul。

```
# 删除Consul中所有的键值
$ curl -X DELETE http://localhost:8500/v1/kv/?recurse
```

```
# 对程序进行编译
$ go build main.go
# 运行它
$ ./main
# Spew is dumping the result for us in details
([]*store.KVPair) (len=1 cap=2) {
(*store.KVPair)(0x10e00de0)({
 Key: (string) (len=27) "docker/nodes/127.0.0.1:2375",
 Value: ([]uint8) (len=14 cap=15) {
  00000000  31 32 37 2e 30 2e 30 2e  31 3a 32 33 37 35
  |127.0.0.1:2375|
 },
 LastIndex: (uint64) 736745
})
```

你也能用 `curl` 来测试获取某一个设置的值。你设置的值也应该能获取到。我们应该继续尝试使用 libkv API 的其他方法，我们把这个留给读者自己来练习。

本章小结

本章讲到了 Libnetwork，它是 Docker Swarm 中最重要的部分之一。我们讨论了它的管理面板、控制面板和数据面板。同时，本章也讲解了如何使用 `libkv`，它是一个键值的抽象，你可以用它来实现自己的发现系统。在下一章中，我们将会学习如何对一个 Swarm 的集群进行安全加固。

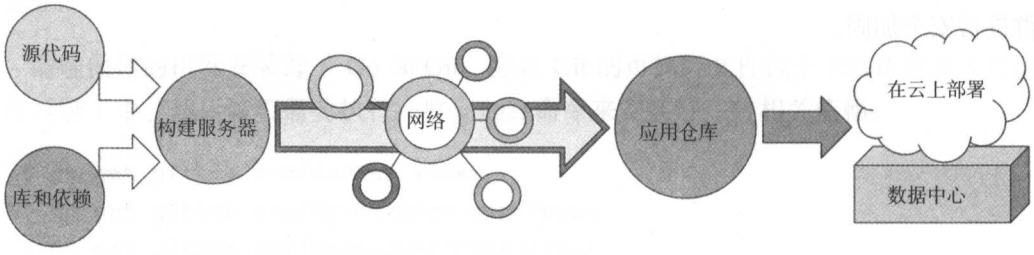
第 9 章

Swarm 集群和 Docker 软件供应链的安全加固

本章将讨论 Swarm 集群安全的相关内容，并会特别关注下面这些主题：

- Docker 的软件供应链（Software supply chain）
- Swarm 集群安全加固的一些推荐做法
- 使用 Docker Notary 来对软件供应链安全加固

软件供应链



Docker 编排只是很大的软件供应链的一个组件。源代码（*Source Code*）作为原始材料，供应链基本上从它开始。源代码会和库与依赖包（*Library and Dependent packages*）整个一起进行编译和连接。构建服务（*Build Service*）用来持续地与源代码和依赖进行集成，并最终把它们组装成一个产品（*Product*）。然后产品通过互联网交付，它会被保存到其他某个类似库房的地方。我们通常把这个库房叫作应用仓库（*Application Repository*），或者

简单地叫作仓库（*Repository*）。最后，产品会送到客户的环境，例如一个位于云上的或物理的数据中心。

使用 Docker 能完美地实现这个工作流。开发者在本地使用 Docker 来编译和测试应用，系统管理员在构建服务上使用 Docker 部署这些应用，同时 Docker 在持续集成的过程中也可能扮演重要角色。

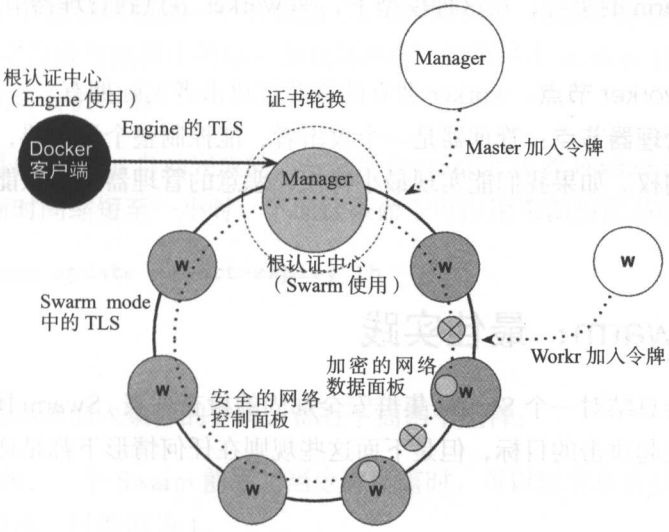
在这里安全很重要。在把产品推送到应用仓库之前，需要有一个安全的方式来对产品进行签名。在以 Docker 为中心的世界里，准备就绪的产品被放在一个叫作 Docker 注册表（Docker Registry）的仓库里。然后，在它每一次被部署到我们运行着 Docker Swarm Mode 的生产环境集群之前，产品的签名都会被校验。

在本章接下来的部分，我们将会讨论两个与安全相关的主题：

- 如何安全加固一个生产环境的 Swarm 集群，和它相关的最佳实践。
- 如何安全加固一个供应链，和 Docker Notary 的使用。

Swarm 集群的安全加固

回忆一下第 4 章“创建生产级别 Swarm”中的那张关于一个安全 Swarm 集群的图，我们将会对 Docker Swarm Mode 集群中的安全点进行解释。



Docker Swarm 的管理器的主要部分之一是调度器（Orchestrator）。Diogo Monica 是一名 Docker 安全组成员，在 2016 年的“最小特权化的调度（Orchestration Least Privileged）”演讲中，其提到了调度器中的每一个组件做事的能力都必须有一个限制。

- **节点管理**：集群的操作者可能会让调度器来对一组节点进行操作。
- **任务分配**：调度器也负责将任务指派到每一个节点。
- **集群状态的调和（reconciliation）**：通过将每一个状态调和到想要的状态，调度器负责维持集群的状态。
- **资源管理**：对于提交的任务，调度器负责资源的提供和回收。

一个只拥有最小特权的调度器将会使系统变得安全，然后一个最小特权化的调度器是基于这些功能来定义的。遵照最小特权原则，一个管理器或者 worker 只能访问完成指定任务必要的信息和资源。

同时，Diogo 在演讲中也讲述了下面五种不同针对 Docker 的攻击模型。它们按照危险系数从低到高排列：

- **外部的攻击者**：来自防火墙之外，试图对集群进行攻击。
- **内部攻击者**：不拥有交换机，但能访问到交换机。可以发送数据包来和集群中的节点进行通信。
- **中间人**：能够监听网络中所有发生的事情，并主动发起攻击的攻击者。如果存在一个 Swarm 的集群，在这种模型下，那 worker 节点到管理器节点的通信会被拦截。
- **恶意的 worker 节点**：worker 拥有的资源被攻击者实际拥有。
- **恶意的管理器节点**：管理器是一个攻击者，能控制整个调度器，并且获得所有资源的访问权。如果我们能实现最小特权，恶意的管理器节点只能攻击与之关联的 worker。

安全加固 Swarm：最佳实践

我们现在将会总结对一个 Swarm 集群安全加固的检查列表。Swarm 团队正在努力达到防止针对整个栈发起攻击的目标，但是下面这些规则在任何情形下都是适用的：

证书颁发机构

第一个保证安全性的重要步骤是决定如何使用证书颁发机构（Certificate Authorities，简称为 CA）。当你使用一个节点来组成一个集群的时候，第一个节点将会自动地为整个集群创建一个自签名的 CA。在它（第一个节点）启动的时候，它会创建好 CA，自己对证书进行签名，为管理器添加证书，然后成为一个准备就绪的单节点集群。当一个新的节点加入的时候，它需要通过提供正确的令牌来获得证书。每一个节点都有自己密码签名的身份标识。同时，系统对于每一个角色——worker 或者管理器都有一个证书。角色位于身份信息之中，用来告知这个节点是谁。如果一个管理器泄露了根 CA，整个集群将会处于危险之中。Docker Swarm Mode 支持使用外部 CA 来维护管理器的身份。管理器可以简单地将 CSR（证书签名请求）转发到外部的 CA，这样便无须维护自己的 CA。请注意，目前唯一支持的协议是 cfssl。下面这个命令是用外部的 CA 来初始化一个集群的：

```
$ docker swarm init --external-ca \
protocol=cfssl,url=https://ca.example.com
```

证书和相互 TLS

在默认情况下，网络控制面板中的每一个端点在通信时都必须使用相互 TLS（Mutual TLS），其是被加密和认证过的。这意味着，worker 无法冒充成管理器，外部的攻击者也无法连接到一个端点然后成功完成 TLS 握手，这是因为攻击者没能双边认证自己的密钥。这也意味着，每一个节点必须提供一个有效的 CA 签名证书，其包含的 OU（Organizational Unit，组织单位）字段能与集群中的每一条规则匹配。如果一个 worker 连接到一个 manger 的端点，它的访问将会被拒绝。

证书的更新是由 Swarm 自动完成的。不论是 SwarmKit 或者 Docker Swarm 模式，你都可以让证书的更新时间缩短至一小时。下面这条命令可以用来调整证书的过期时间：

```
$ docker swarm update --cert-expiry 1h
```

集群加入令牌

每一个被节点用来加入集群的令牌，都有下面四个组件：

- SWMTKN，一个 Swarm 前缀，当令牌泄露时，可以用它来查找定位和过滤。
- 令牌的版本，目前值为 1。

- CA 根证书的密码散列值，可以用来 bootstrap。
- 一个*随机*生成的 Secret。

下面是一个令牌的例子：

```
SWMTKN-1-11lo1xx5bau6nmv5jox26rc5mr7l1mj5wi7b84w27v774frtko-e82x3ti068m9
eec9w7q2zp9fe
```

要访问集群，有必要发送令牌作为证明。这就像一个集群的密码。

好的消息是，如果令牌泄露，可以简单地轮换（rotate）掉令牌：

```
$ docker swarm join-token --rotate worker
$ docker swarm join-token --rotate manager
```

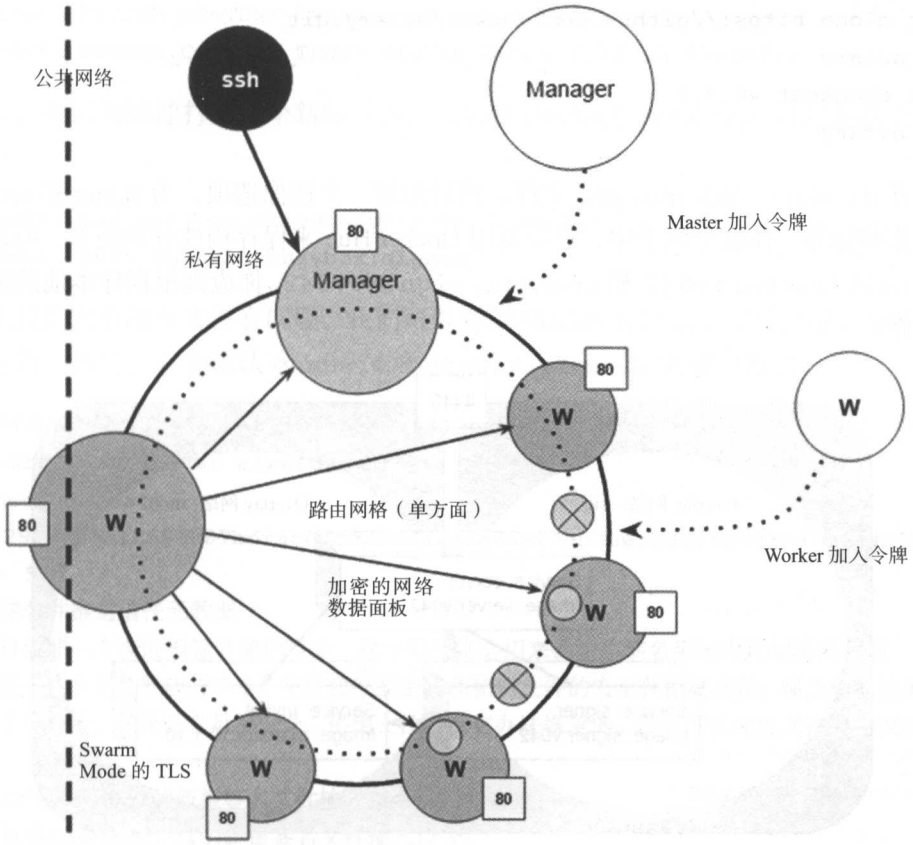
在 Docker Machine 中添加 TLS

另外一个好的实践是，通过 Docker Machine 来配置所有的管理器节点，它会自动地设置额外的 TLS 层，这样每一个管理器都能被远端的 Docker 客户端安全地访问。这可以通过下面这条命令办到，与之前章节中的做法类似：

```
$ docker-machine create \
  --driver generic \
  --generic-ip-address=<IP> \
mg0
```

在私有网络上组成一个集群

如果没有组建一个混合式集群的必要，那么一个最好的实践是组成一个所有节点都在本地私有网络中的集群。这样，将不必对 overlay 网络的数据加密，集群也能获得更好的性能。在组成一个这种类型的集群的时候，路由网格（Routing Mesh）允许我们把任何一个 worker，不一定是管理器，暴露到公共网络的接口。下面这张图展示了这种配置的集群。从图中可以看到，在这种配置中，Docker 的服务发布在入口（Ingress）网络的端口 80。路由网格组成了一个星状网格，但是我们将它进行了简化，只显示了大 W 节点的一侧，这一侧连接了 IPVS（IP 虚拟服务器）并负责将负载均衡到其他节点。大 W 节点有两个网络接口。它的公共接口允许把节点变成整个集群的前端节点。在这种架构中，我们可以通过避免将管理器节点暴露在公共网络上来取得一定程度的安全性。



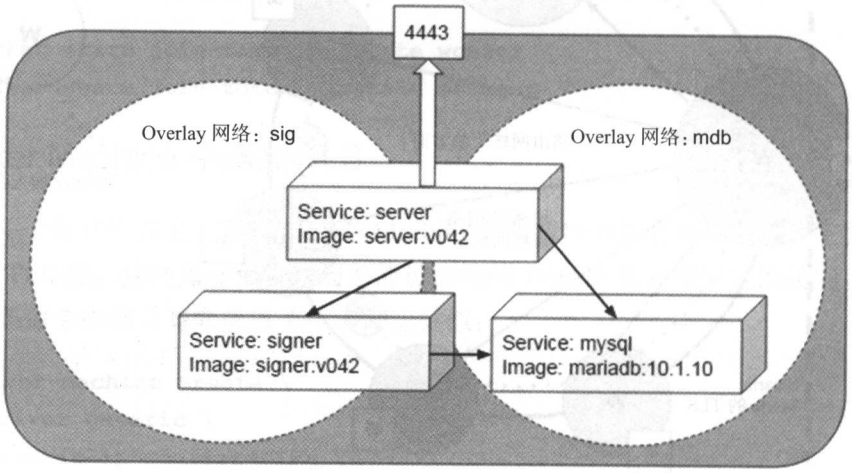
Docker Notary

Docker 内容信任机制使用 Docker Notary（https://github.com/docker/notary）来实现。Notary 是一个基于 The Update Framework（简称为 TUF，https://github.com/theupdateframework/tuf）的项目。TUF 作为一个安全框架，能让我们一次传递一组信任的内容。Notary 简化了内容的发布和验证，能让客户端和服务端组成一个可信任集合（trusted collection）。如果我们有一个 Docker 镜像，我们可以离线地使用一个安全的线下秘钥进行签名。然后，当该镜像要发布的时候，可以将它推送到一个用来发布可信任镜像的 Notary 服务器。对于使用 Docker 的企业来说，Notary 是一个可以用来实现安全软件供应链的方式。

第一步是复制一份 Notary 仓库的代码（在这个例子中，我们是使用的版本为 0.4.2）：

```
git clone https://github.com/docker/notary.git
cd notary
git checkout v0.4.2
cd notary
```

打开 docker-compose.yml 文件，然后添加一个镜像选项，为 signer 和 server 指定镜像名称和标签。在这个例子中，笔者使用 Docker Hub 来保存构建好的镜像。所以，那就是 chanwit/server:v042 和 chanwit/signer:v042。你应该根据你本地的情况修改配置文件。



然后，使用下边的命令进行启动：

```
$ docker-compose up -d
```

Notary 服务器启动并运行起来后，其监听地址为：https://127.0.0.1:4443/。要让 Docker 客户端可以和 Notary 完成 TLS 握手，我们需要将 Notary 服务器证书复制一份作为这个受信任地址（127.0.0.1:4443）的 CA：

```
$ mkdir -p ~/.docker/tls/127.0.0.1:4443/
$ cp ./fixtures/notary-server.crt
~/.docker/tls/127.0.0.1:4443/ca.crt
```

然后，我们启用 Docker Content Trust（Docker 内容信任），然后将 Docker Content Trust 服务器指向我们运行在 https://127.0.0.1:4443 的 Notary。

```
$ export DOCKER_CONTENT_TRUST=1
$ export DOCKER_CONTENT_TRUST_SERVER=https://127.0.0.1:4443
```

然后，我们为镜像打上一个新的标签，然后在 Docker Content Trust 启用的状态下推送镜像。

```
$ docker tag busybox chanwit/busybox:signed
$ docker push chanwit/busybox:signed
```

如果我们之前的配置没有问题，我们可以看到 Docker 客户端会需要新的 root key 和新的仓库秘钥。然后，它会确认 `chanwit/busybox:signed` 被成功签名。

```
推送涉及一个仓库，[docker.io/chanwit/busybox]
e88b3f82283b: Layer already exists
signed: digest:
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
  size: 527
正在签名并推送信任元数据
你即将创建一个新的根签名秘钥口令。这个口令将会用来保护你签名系统中最敏感的秘钥
请选择一个长的、复杂的口令，并注意保证密码和秘钥文件的安全并做好备份。强烈建议使用密码管理工具来生成口令，并保证它足够安全。秘钥文件是没有办法恢复的。你可以在配置文件夹中找到这个秘钥文件
请为ID为1bec0c1的新根签名文件输入口令：
请重复为ID为1bec0c1的新根签名文件输入口令：
输入新的仓库秘钥文件的口令，其ID号为ee73739：
重复输入新的仓库秘钥文件的口令，其ID 为ee73739 (docker.io/chanwit/busybox)的口令：)
已完成初始化"docker.io/chanwit/busybox"
已成功签名"docker.io/chanwit/busybox":signed
```

现在，我们可以尝试拉取同一个镜像：

```
$ docker pull chanwit/busybox:signed
Pull (1 of 1):
chanwit/busybox:signed@sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec
43daf19026a7ce69912
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912:
Pulling from chanwit/busybox
Digest:
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
```

```
   Status: Image is up to date for
 chanwit/busybox@sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19
026a7ce69912
   Tagging
chanwit/busybox@sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19
026a7ce69912 as chanwit/busybox:signed
```

当我们在拉取一个没有被签名的镜像的时候，我们可以发现没有信任的数据存在：

```
   $ docker pull busybox:latest
   Error: remote trust data does not exist for docker.io/library/busybox:
127.0.0.1:4443 does not have trust data for docker.io/library/busybox
```

Docker Secret 介绍

Docker 1.13 版本在 Swarm 中包含了一个新的概念，叫作 secret 管理（译注 secret 指证书、密码、令牌和其他敏感信息）。

我们知道，Swarm mode 需要使用各种 secret。在我们初始化 Swarm 的时候，Swarm 会为我们生成一些秘钥。

```
$ docker swarm init
```

Docker 1.13 版本通过一个新的命令来增加了 secret 的管理，其目的是有效地对 secret 进行处理。secret 相关的子命令有：`create`、`ls`、`inspect` 和 `run`。

让我们创建我们的第一个 secret。`secret create` 子命令接受一个来自标准输入的 secret，所以我们可以输入我们的秘钥，并按下 Ctrl+D 组合键来保存内容。注意不要按下 Enter 键。我们需要的密码是 `1234`，而不是 `1234\n`，例如：

```
$ docker secret create password
1234
```

然后，两次按下 Ctrl + D 组合键来关闭标准输入。

我们可以检查是否有一个名为 password 的 secret 存在：

```
$ docker secret ls
ID                          NAME                CREATED             UPDATED
```

```
16blafexuvrv2hgznrjitj93s  password  25 seconds ago   25 seconds ago
uxep4enknneoevvqatstouec2  test-pass 18 minutes ago   18 minutes ago
```

secret 该如何使用呢？如果在创建 service 的时候指定了 secret 选项，secret 的内容可以和一个 service 绑定。secret 就是一个在/run/secrets/目录下的文件。在我们这里的例子中，在该目录下生成了一个/run/secrets/password 文件，其中包含的内容是字符串 1234。

设计 secret 的目的是取代滥用的环境变量。例如，在 MySQL 或者 MariaDB 容器中，它的 root 用户密码就可以设置成一个 secret，而不用通过环境变量传入。

我们将会展示一个小小的 hack，来让 MariaDB 支持新的 Swarm secret，我们先从获取下面这个 entrypoint.sh 文件开始：

```
$ wget https://raw.githubusercontent.com/docker-
library/mariadb/2538af1bad7f05ac2c23dc6eb35e8cba6356fc43/10.1/docker-
entrypoint.sh
```

在检查 MYSQL_ROOT_PASSWORD 环境变量之前，我们需要在其中加入一行代码，位置大约在第 56 行：

```
# check secret file. if exist, override
if [ -f "/run/secrets/mysql-root-password" ]; then
     MYSQL_ROOT_PASSWORD=$(cat /run/secrets/mysql-root-password)
fi
```

上面这段代码会检查/run/secrets/mysql-root-password 文件是否存在，如果存在，就将 secret 文件的内容赋值给环境变量 MYSQL_ROOT_PASSWORD。

然后，我们就可以创建一个 Dockerfile，并覆盖 MariaDB 镜像中默认使用的 docker-entrypoint.sh。

```
FROM mariadb:10.1.19
RUN  unlink /docker-entrypoint.sh
COPY docker-entrypoint.sh /usr/local/bin/
RUN  chmod +x /usr/local/bin/docker-entrypoint.sh
RUN  ln -s usr/local/bin/docker-entrypoint.sh /
```

然后，让我们来构建镜像：

```
$ docker build -t chanwit/mariadb:10.1.19 .
```

回想一下我们之前创建过一个命名为 password 的 secret，而我们这个镜像要从路径为 /run/secrets/mysql-root-password 的 secret 文件中读取内容并设置 MariaDB 的 root 用户密码。所以，这个镜像希望访问的是/run/secret 目录下的一个文件名不相同的文件。对于这种情况，我们可以使用 secret 完整的选项(source=password, target=mysql-root-password）来让我们的 service 正常工作。例如，我们现在可以启动一个新的命名为 mysql 的 Swarm service，它使用我们新构建的 MariaDB 镜像：

```
$ docker network create -d overlay dbnet
lsc7prijmvg7sj6412b1jnsot
$ docker service create --name mysql \
  --secret source=password,target=mysql-root-password \
  --network dbnet \
  chanwit/mariadb:10.1.19
```

要验证 secret 是否正常工作，我们可以启动一个 PHPMyAdmin 实例，让其运行在同一个 overlay 网络上。不要忘记将这些服务连接起来，并给 myadmin 服务传递-e PMA_HOST=mysql 选项。

```
$ docker service create --name myadmin \
  --network dbnet --publish 8080:80 \
  -e PMA_HOST=mysql \
  phpmyadmin/phpmyadmin
```

然后，你就可以打开浏览器，访问 http://127.0.0.1:8080，使用 root 用户，用密码 1234 登录进 PHPMyAdmin，这个密码是通过 Docker secret 提供的。

本章小结

在本章中，我们了解了如何对 Docker Swarm mode 和 Docker 的软件供应链进行安全加固。我们讨论了在生产环境中对 Docker Swarm 集群进行安全加固的一些最佳实践。然后我们继续讨论了 Notary，它作为一种安全送达机制，可以完成 Docker Content Trust。本章最后，我们对 Docker 的一个新特性——secret 管理，做了大概介绍。我们还展示了如何使用 Docker secret 来部署 MySQL/MariaDB 服务器，一改传统采用环境变量传递 root 用户密码的做法。在下一章中，我们将探索如何在一些公有云上部署 Docker 和 OpenStack。

第 10 章 Swarm 和云

在这本书之前，我们在不同的底层技术上运行 Docker Swarm，到目前为止却尚未深入探讨这个隐含事实：我们可以在 AWS、DigitalOcean 和本地工作站上运行 Swarm。如果仅用于测试或临时目的，那在什么地方运行 Swarm 可能是次要考虑的问题（*让我们启动一些装有 Docker Swarm 的 AWS 实例，只要能工作其余的就随它吧*），但是对于生产环境来说，却很有必要理解每种做法的优缺点，为什么要那么做，并对时下流行的做法进行评估后采用。

在本章中，我们将会讨论一些在公有云和私有云中的做法和相关技术，以及它们的共同点。在第 11 章“Swarm 的未来展望”中我们会来解释时下的热词 **CaaS（容器即服务）**和 **IaaC（基础设施即代码）**。

我们将主要关注以下内容：

- Docker for AWS 和 Docker for Azure
- Docker 数据中心
- OpenStack 上的 Docker

Docker for AWS 和 Docker for Azure

Docker 团队很早就开始打造供运维人员使用的新一代工具集： Docker for AWS 和 Docker for Windows。它们的历史有 Docker for Mac 和 Docker for Windows 那么久。它们意在提供 Docker 基础设施的一种自动化部署体验，特别是能够运行 Swarm 的基础设施。

它们的目标是要提供一种做事的标准方法，让 Docker 工具和底层基础设施集成，让人们能很轻松地在他们热爱的平台上运行自己喜爱软件的最新版本。其最终目标是让他们的项目可以从本地运行着 Docker for Mac 或 Docker for Windows 的笔记本电脑上，移动到运行着 Docker for AWS 或者 Docker for Azure 的云上。

Docker for AWS

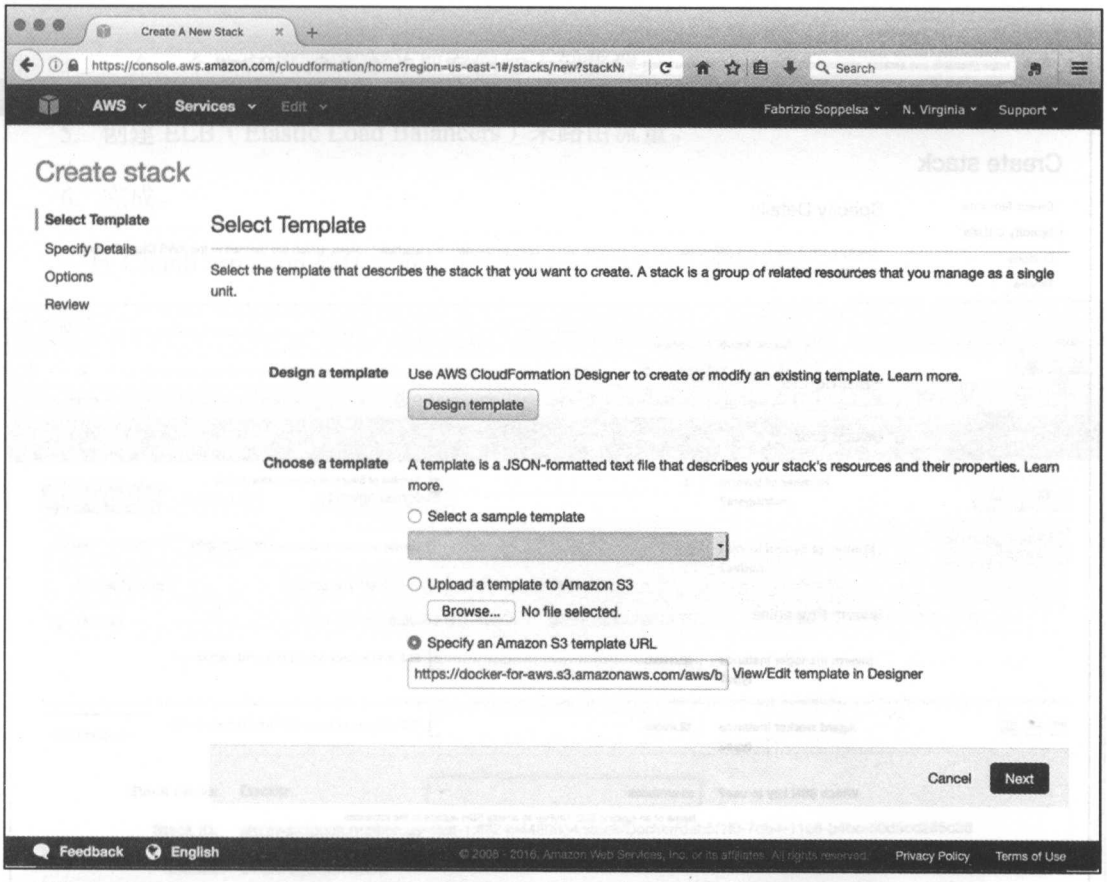
Docker for AWS 的用户体验十分棒，与在 Docker 生态环境中一贯感受到的一样。它的需求是：

- 一个 AWS ID
- 一个导入进 AWS keyring 的 SSH 秘钥
- 准备好的安全组

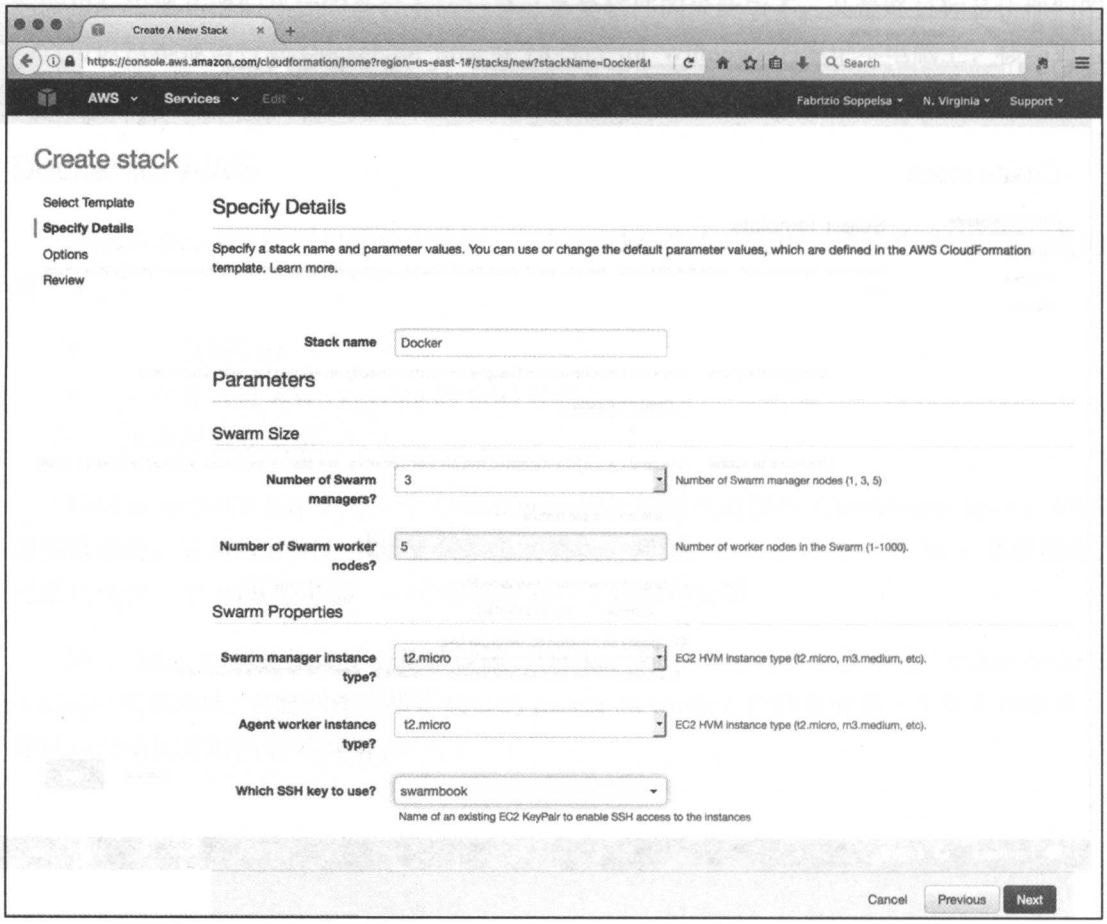
Docker for AWS 基本上是一个 CloudForms 模板的可点击版本。CloudForms 是一个 AWS 的编排系统，它可以让用户创建复杂系统的模板，例如，你可以定义一个 Web 基础设施，要求它包含三个 Web 服务器、一个数据库和一个负载均衡器。

除了 Web 栈和其他常见的基础设施，Docker for AWS 自然也能创建 Docker Swarm（mode）基础设施：它能创建出指定数量的 master 和 worker，在前面放置一个负载均衡器，并且相应地配置好所有的网络。

这是欢迎页面：


Create A New Stack
https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new?stackN
Search
AWS
Services
Edit
Fabrizio Soppelsa
N. Virginia
Support
Create stack
Select Template
Specify Details
Options
Review
Select Template
Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.
Design a template
Use AWS CloudFormation Designer to create or modify an existing template. Learn more.
Design template
Choose a template
A template is a JSON-formatted text file that describes your stack's resources and their properties. Learn more.
Select a sample template
Upload a template to Amazon S3
Browse...
No file selected.
Specify an Amazon S3 template URL
https://docker-for-aws.s3.amazonaws.com/aws/b
View/Edit template in Designer
Cancel
Next
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

然后，你可以对一些基础的和高级的选项进行设定：



可以看到，我们可以设定管理器和 worker 的数量，以及要启动的实例的配置类型。目前，它支持多达 1000 个 worker。然后，只需要在下一步中点击“Create Stack（创建栈）”即可，然后稍微等待几分钟，CloudForms 就能启动好整个基础设施。

模板会做如下的事情：

1. 在 AWS 账户下创建一个新的 Virtual Private Cloud、网络和包含的子网。

2. 创建两个 Auto Scaling 组，分别供管理器和 worker 使用。

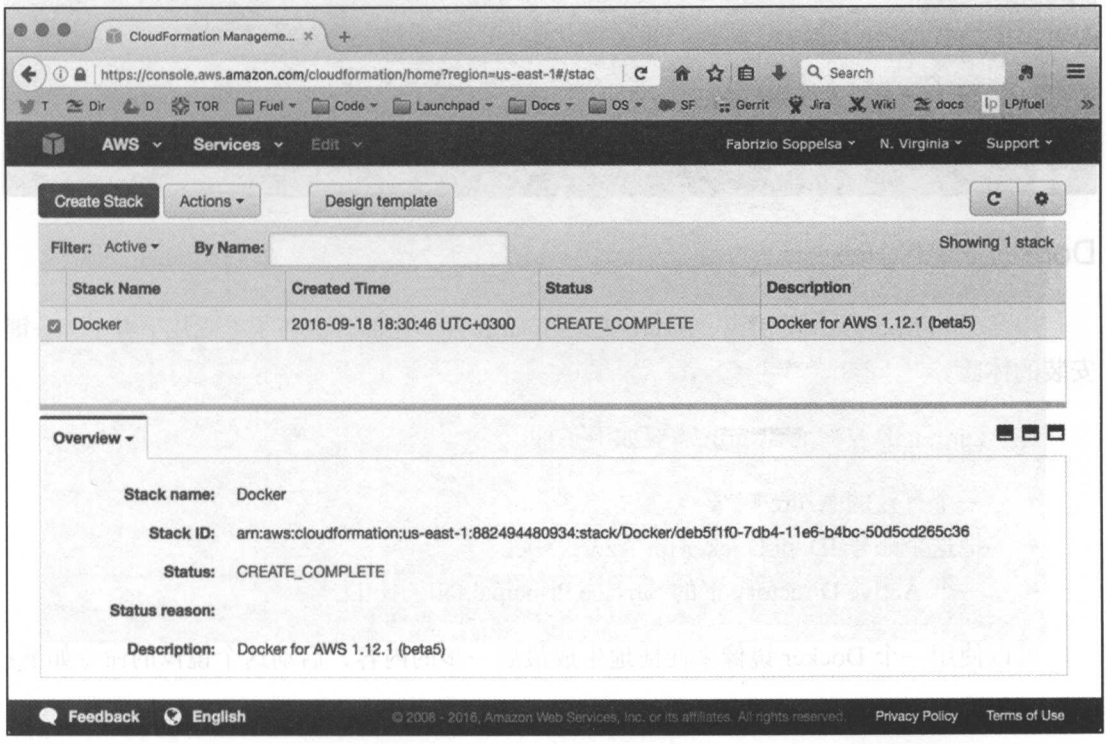
3. 启动所有管理器，并保证它们处于健康的启动状态，并满足了 Raft 协议 quorum 的要求。

4. 一个一个地启动 worker 并把它们加入到集群。

5. 创建 ELB（Elastic Load Balancers）来路由流量。

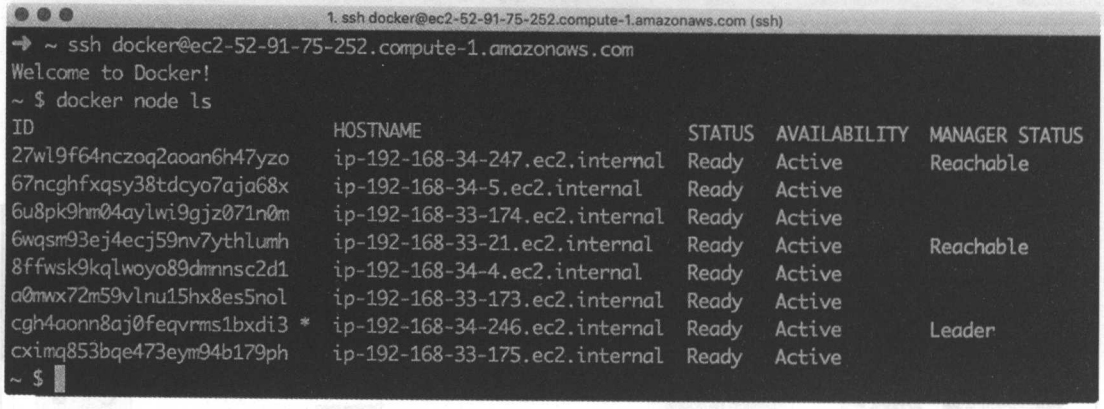
6. 完成。

一旦 CloudFormation 操作完成，它会弹出一个绿色的确认信息。



现在，我们可以进入到 Docker Swarm 基础设施中。可以随便选择一个 manager 的公网 IP 地址，然后使用在第一步指定的 SSH 秘钥进行登录：

```
ssh docker @ec2-52-91-75-252.compute-1.amazonaws.com
```



Docker for Azure

得益于和 Microsoft 的协议，Azure 上的 Swarm 自动化部署也有了（或几乎有了）一键安装的体验。

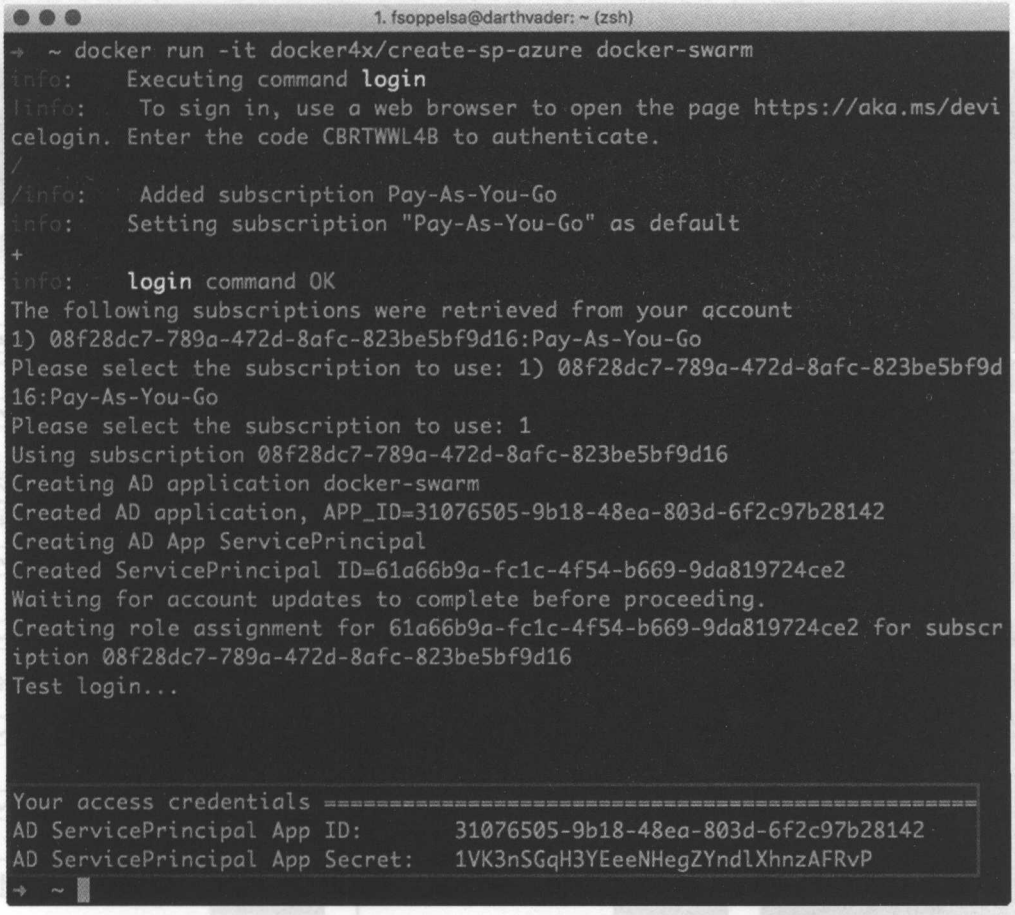
在 Azure 上部署一个 Swarm 需要如下条件：

- 一个有效的 Azure 账号。
- 把这个账号 ID 和 Docker for Azure 关联。
- 一个 Active Directory 下的 Service Principal 的应用 ID。

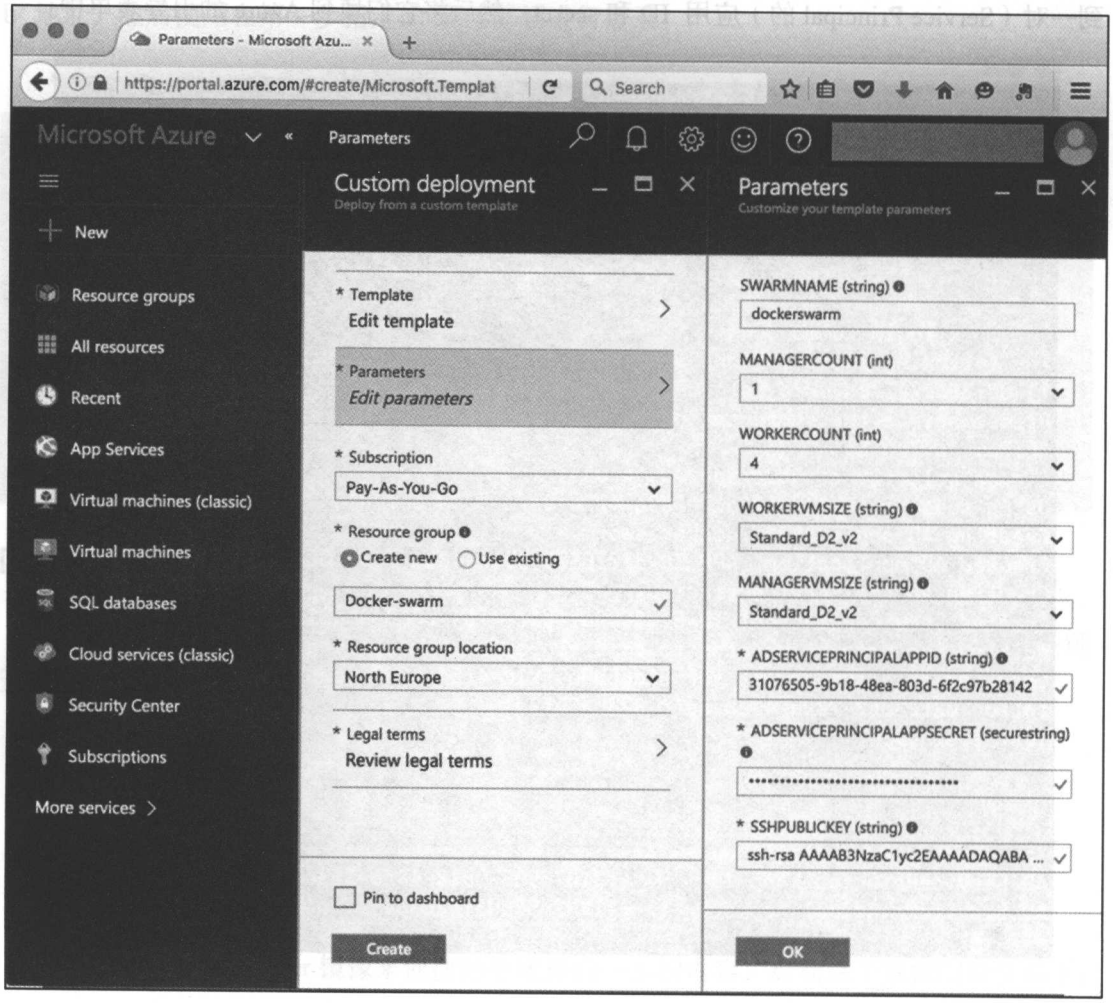
可以使用一个 Docker 镜像来便捷地生成最后一步的内容，启动这个镜像的命令如下：

```
docker run -it docker4x/create-sp-azure docker-swarm
```

在这个镜像的运行过程中，会要求通过浏览器访问某个 URL 并登录。最后，我们会拿到一对（Service Principal 的）应用 ID 和 secret，然后将它们填到 Azure 的引导表单中：

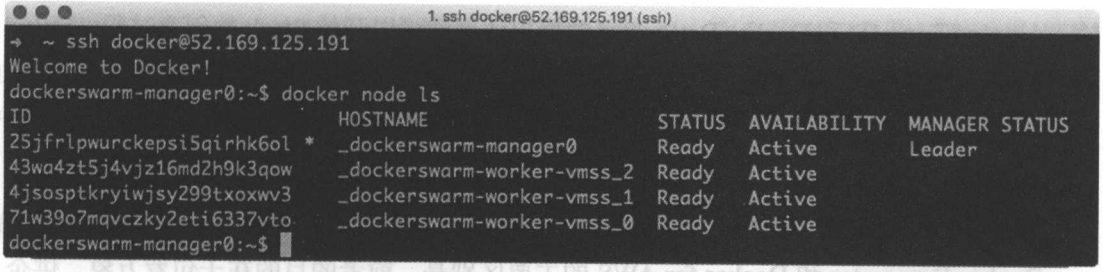


在一切就绪之后，单击 OK 和 Create 按钮。



然后，它会创建一组传统的虚拟机，来运行设定数量的管理器（本例中是 1）和 worker（本例中是 4），同时它也会一并将内部网络、负载均衡器和路由器配置好。同 Docker for AWS 一样，我们可以通过登录到一个管理器的公网 IP 使用部署好的 Swarm。

```
ssh docker@52.169.125.191
```



目前 Azure 模板有一个局限，它只支持一个管理器，不过它应该很快就能支持添加新的管理器。

Docker Datacenter

Docker Datacenter（DDC），即之前被 Docker 收购的 Tutum，可以让我们一键部署并使用 UCP（Universal Control Panel，通用控制面板）。UCP 是 Docker 的一款面向企业的商业产品。

Docker Datacenter 包含：

- Universal Control Panel（UCP，通用控制面板），它是 DDC 的用户界面，可以参考 https://docs.docker.com/ucp/overview。
- Docker Trusted Registry（DTR，Docker 信任注册表），它是 DDC 的私有注册表，可以参考 https://docs.docker.com/docker-trusted-registry。

在 Dockercon 16 上，Docker 团队发布了 Docker Datacenter，对 AWS 和 Azure 支持（目前处于 Beta 阶段）。要试用 Docker Datacenter，你需要将一个授权证书和公司或项目的 AWS/Azure ID 关联起来。

如同 Docker for AWS 一样，Datacenter for AWS 有一个 CloudFormation 的模板可以让我们能立刻启动好一个 Docker Datacenter。其必要条件是：

- 至少要配置好 Router53，即 AWS DNS 服务，可以参考：http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html。
- 一个 Docker Datacenter 的授权证书。

我们需要做的是访问证书中的链接，然后进入 Create Stack 页面。在该页面上，只需要

输入 HostedZone ID 和 Docker Datacenter 的授权证书就可以开始 Stack 的创建过程。Docker Datacenter 在幕后会把一些 VM 放在一个私有的网络中(nodes),然后在另外一些拥有 Elastic Load Balancer 负载均衡（ELBs，供 controller 使用）的 VM 上安装引擎的商业支持版本。目前 Docker Datacenter 版本的 VM 在内部运行的是 Swarm 单节点集群，且包含一个便于彼此连接的发现机制。我们可以期待 Datacenter 稳定版本会很快发布。

Docker Datacenter 和 Docker for AWS 的主要区别是，前者的目的在于包罗万象，供企业使用。尽管后者的方案对于部署 Swarm 集群这个特定场景来说速度更快，前者却是一个更加完备的解决方案，它有一个吸引人的用户界面，并包含 Notary 和其他一些生态中可选的服务。

OpenStack 上的 Swarm

说到私有云，最流行的开源 IaaS 解决方案自然是 OpenStack。OpenStack 是一个伟大的 program（之前称作 project）生态系统，其目的是提供一个云操作系统。OpenStack 的核心 program 包括：

- Keystone：身份和授权的系统。
- Nova：虚拟机抽象层。Nova 可以插入虚拟化模块，如 Libvirt，VMware。
- Neutron：网络模块，负责处理租户的网络、实例的端口、路由和流量。
- Cinder：存储模块，负责处理 volume。
- Glance：图片存储。

它们的连接都通过外部的角色完成：

- 一个数据库系统，如 MySQL，负责配置的保存。
- 一个 AMQP 的代理，如 Rabbit，负责将操作进行排队和送达。
- 一个代理系统，如 HAproxy，负责代理 HTTP API 请求。

在 OpenStack 中一个典型的虚拟机创建过程，会有下面这些步骤：

1. 用户决定要启动一个 VM，并打开 UI（Horizon）或者命令行。
2. 用户单击 UI 上的一个按钮，或者在命令行中输入一条命令如 `nova boot……`

3. Keystone 会在它的租户中检查认证并核实授权，它会检查用户数据库或者查询 LDAP（具体情况依 OpenStack 配置而定），然后生成一个令牌，该令牌可以在整个会话中使用："这是你的令牌：gAAAAABX78ldEiY2"。

4. 如果认证通过，并且用户拥有启动 VM 的授权，会使用户的令牌调用 Nova，"我们想要启动一个 VM，请找一个合适的能放置该 VM 的物理主机"。

5. 如果找到了这样一个主机，Nova 会从 Glance 获取用户选择的镜像："Glance，请给我一个 Ubuntu Xenial 的可启动 qcow2 的文件"。

6. 在要物理启动的虚拟机的计算主机（compute host）上，有一个叫作 nova-compute 的进程会和配置好的插件（如 Libvirt）进行交流："我们要在这个主机上启动一个虚拟机"。

7. Neutron 会为 VM 分配私有的（如有必要，公有的）网络端口："请在这些子网池（subnet pool）中，在指定的网络上创建这些端口"。

8. 如果有需要，Cinder 会在调度器指定的主机上分配 volume。"让我们创建一些额外的卷，然后连接到 VM 上"。

9. 如果使用了 KVM，会生成一个包含所有上述信息的 XML 文件，然后 Libvirt 会在计算机主机上启动 VM。

10. 在 VM 启动后，可以通过 cloud-init 注入变量，如允许无密码登录的 SSH 秘钥。

而这些步骤（除了在 Cinder 上的第 8 步），也正是 Docker Machine 的 OpenStack 驱动所做的事情：当你用 Machine 创建一个 Docker Host，并使用-d openstack 选项（指定使用 OpenStack 驱动）的时候，你必须指定一个已经存在的 glance 镜像和一个已经存在的私有网络（也可以选用公有网络），然后指定一个保存在 Nova 数据库里面的 SSH 镜像（可选，不指定会自动生成）。必须给 Machine 传入一些授权相关的环境变量到 OpenStack 环境中，或者你也可以采用从文件导出 shell 环境变量的方式。

下面这条命令用来在 OpenStack 上创建一个 Docker Host：

```
docker-machine create \
--driver openstack \
--openstack-image-id 98011e9a-fc46-45b6-ab2c-cf6c43263a22 \
--openstack-flavor-id 3 \
```

```
--openstack-floatingip-pool public \
--openstack-net-id 44ead515-da4b-443b-85cc-a5d13e06ddc85 \
--openstack-sec-groups machine \
--openstack-ssh-user ubuntu \
ubuntu1
```

OpenStack Nova

在 OpenStack 上使用 Docker Swarm 的传统做法，首先是在专用的网络上创建一些实例，如 10 个使用 Ubuntu 16.04 镜像的 VM。我们可以：

- 在 Web 界面上，指定实例的数目为 10 。
- 或者通过命令行，使用 `nova boot ... --max-count 10 machine-`。
- 或者使用 Docker Machine。

最后一种做法更加有前景，因为 Machine 会自动安装 Docker，不需要在稍后进行一些 hack 或者在新创建的实例上使用其他工具（如使用 Machine 的 generic 驱动、belt、Ansible、Saltstack 或其他脚本）。但是在撰写本书的时候，Machine 还不支持创建批量主机（bulk-host），所以你需要使用一些基本的 shell，循环调用 `docker-machine` 命令：

```
#!/bin/bash
for i in `seq 0 9`; do
    docker-machine create -d openstack ... openstack-machine-$i
done
```

这种用户体验一点也不好，除此之外当到了几十个主机的规模时，Machine 的伸缩能力仍然比较差。

已成过去：废弃的 nova-docker 驱动

曾经，有一个 Nova 的驱动，它能插入 Docker 容器并把其作为 Nova 的最终目的地（这些驱动能允许从 Nova 中创建并且管理 Docker 容器，而不需要创建 KVM 或者 VmWare 的 VM）。如果是针对老版本的 Swarm，使用这个工具那还可以理解（因为编排的一切都是以容器的方式），但这对于 Swarm Mode 来说没有什么意义，因为它需要 Docker Host 而不是简单的容器。

当下现实：OpenStack 友好的方式

幸运的是，OpenStack 是一个非常有生命力的项目，现在已经发到了版本 O（Ocata），它能通过很多可选的模块来增强。从 Docker Swarm 的角度，最有意思的模块是：

- Heat：这是一个调度系统，可以从模板中创建 VM 配置。
- Murano：这是一个应用目录（catalog），可以从一个开源社区维护的目录中运行应用，包括 Docker 和 Kubernetes 的容器。
- Magnum：这是一个 Rackspace 提供的容器，即服务方案。
- Kuryr：这是一个网络的抽象器。有了 Kuryr，你可以把 Neutron 的租户网络和用 Docker Libnetwork 创建的 Docker 网络连接起来（如 Swarm 的网络），然后把 OpenStack 的实例和 Docker 的容器连接起来，就好像它们连上了同一个网络。

OpenStack Heat

OpenStack Heat 有一点像 Docker Composer，它也能让你通过一个模板来启动系统，但是它更加强大：你不仅仅能从一个镜像启动一组实例，例如 Ubuntu 16.04 镜像，你也能够对它们进行编排，这意味你可以创建网络，将一些 VM 的接口连接到网络，放置负载均衡器或在实例上执行后续任务，例如安装 Docker。Heat 在 OpenStack 中的角色基本上相当于亚马逊的 CloudFormation。

在 Heat 中，YAML 模板是一切的基础，在启动之前，你可以用它来以对基础设施进行建模，就像使用 Compose 一样。例如，你可以创建一个像这样的模板文件：

```
...
resources:
  dockerhosts_group:
    type: OS::Heat::ResourceGroup
    properties:
      count: 10
      resource_def:
        type: OS::Nova::Server
        properties:
          # create a unique name for each server
          # using its index in the group
          name: docker_host_%index%
```

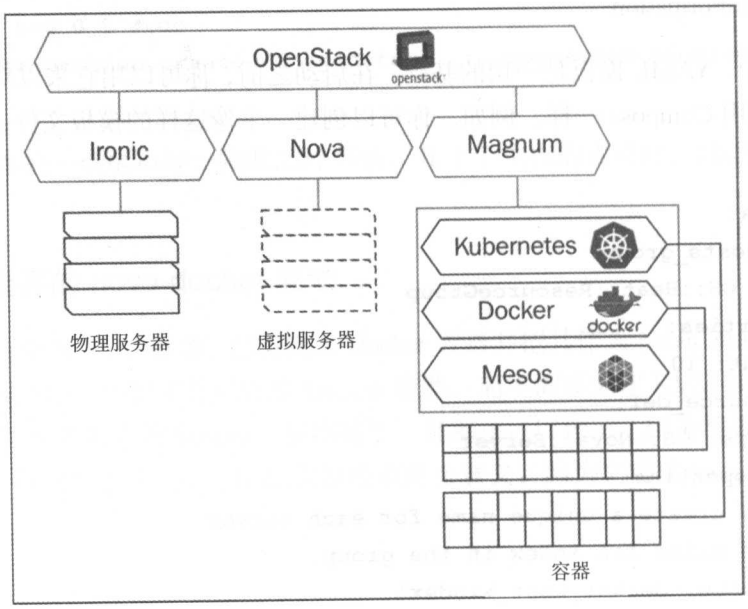
```
        image: Ubuntu 16.04
        flavor: m.large
        image: Ubuntu 16.04
        flavor: m.large
    ...
```

然后你可以用它启动一个 stack(`heat stack-create -f configuration.hot`)。Heat 会调用 Nova、Neutron、Cinder 和所有必要的 OpenStack 服务，来对资源进行编排并让它们可用。

这里我们不打算展示如何通过 Heat 来启动一个 Docker Swarm 的基础设施，我们打算介绍一下 Magnum，它在底层使用 Heat 来操作 OpenStack 的对象。

OpenStack Magnum

Magnum，发布于 2015 年年末，由 OpenStack Containers 团队开发，目标是让容器编排引擎（Container Orchestration Engine，简称 COE），如 Docker Swarm 和 Kubernetes 可以在 OpenStack 中作为第一类的资源。在 OpenStack 过去有并以后也会有众多专注于提供集群支持的项目，但是 Magnum 走得更远，因为它是设计来支持容器编排的，而不仅限于容器管理。



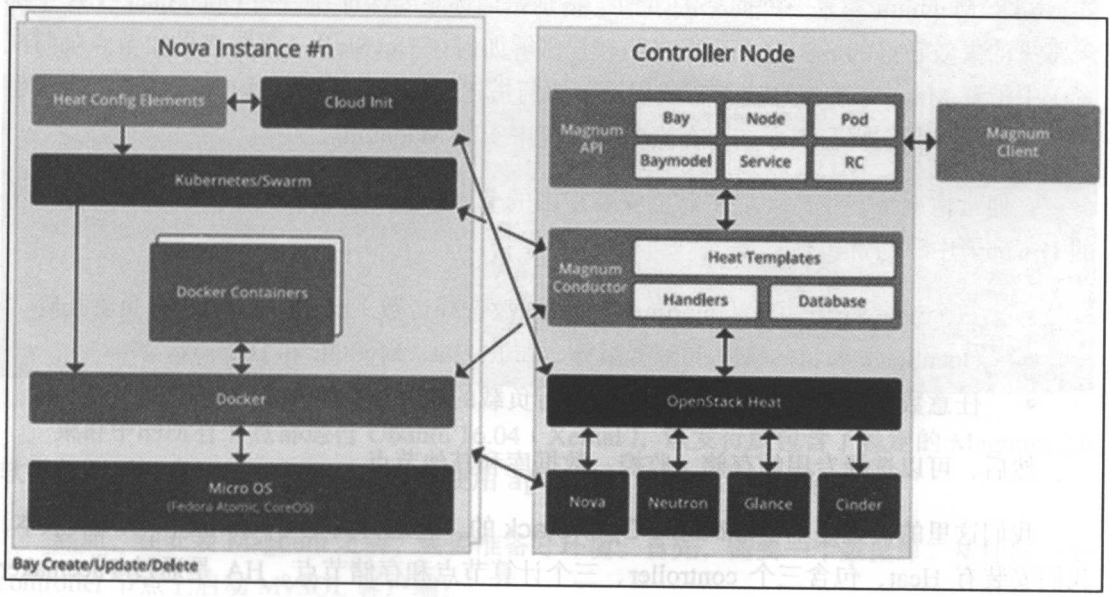
到目前为止，人们都特别地关注 Kubernetes，但是为什么我们要在这里讨论 Magnum 呢？因为它是一种在私有云中便捷地进行 CaaS 编排的最有前景的开源技术。Magnum 不支持最新的 Swarm Mode，至少在撰写本书的时候还不支持：这个需求必须被解决。有一本由作者写的 Launchpad 蓝皮书，最终它有可能在本书发布之后开始准备：https://blueprints.launchpad.net/magnum/+spec/swarm-mode-support。

架构和核心概念

在 Magnum 的控制节点上，运行着两个主要组件：

```
magnum-api
magnum-conductor
```

第一个进程，`magnum-api` 是一个典型的 OpenStack API 提供者， Magnum 的 Python 客户端或其他进程调用它来执行操作，如创建一个集群。而 `magnum-conductor` 是通过 AMQP 服务器（如 Rabbit）被 `magnum-api` 调用（它多多少少与 `nova-conductor` 的功能相同），其目标是作为 Kubernetes 或者 Docker API 的接口。在实践中，这两个二进制文件会一起运行并提供一种调度抽象。



在 OpenStack 集群的计算节点上，除了 `nova-compute` 进程，并不需要运行什么特别

的东西：Magnum conductor 会直接利用 Heat 来创建 stack，然后在 Nova 中创建网络并且实例化 VM。

Magnum 的一些概念随着项目的发展也在不断变化。下面是一些主要的概念：

- Container 指的是 Docker 容器。
- 一个 Cluster（之前叫 Bay）是一个节点对象的集合，工作会调度到上面，如 Swarm 节点。
- 一个 ClusterTemplate（之前叫 BayModel）是一个保存集群类型信息的模板。例如，一个 ClusterTemplate 定义了一个包含三个管理器和五个 worker 的 Swarm 集群。
- Pod 是一个运行在同一个物理主机或者虚拟主机上的容器的集合。

Magnum 是一个非常活跃的项目，对于高级的功能，如存储、对新 COE（容器编排引擎）的支持和伸缩，我们可以关注它的发展：http://docs.openstack.org/developer/magnum/。

Mirantis OpenStack 上 HA Magnum 的安装

安装 Magnum 不是一件简单的事情，特别是当你想要保证在一些 OpenStack HA 中部署常见的失效转移的话。网上有很多的教程讲解如何在 DevStack（开发者的单节点临时配置）中配置 Magnum，但是还没有教程展示如何搭建一个包含超过一个 controller 的真实生产环境。这里我们将会展示如何在真实的环境中安装 Magnum。

在通常情况下，生产环境中需要安装很多的节点来满足各种不同的需求。在一个最小的 HA 部署中，通常包括：

- 三个或者多个（因为 quorum 机制的缘故选择奇数）**controller 节点**，负责托管 OpenStack 项目的 API 和配置服务，如 Rabbit、MySQL 和 HAproxy。
- 任意数量的**计算节点**，作为物理运行负载的地方（VM 放置的地方）。

然后，可以选择专用的存储、监控、数据库和其他节点。

我们这里的设置是基于 Mirantis OpenStack 的，它运行的是 OpenStack Newton 版本，我们安装有 Heat，包含三个 controller、三个计算节点和存储节点。HA 是通过 Pacemaker 来配置的，它会让如 MySQL、RabbitMQ 和 HAproxy 等资源处于高可用的状态。这里有一个截图显示配置到 Pacemaker 的资源，它们都已经启动并且处于正常运转的状态。

```
1. root@node-2: ~ (ssh)
root@node-2:~# pcs resource
 Clone Set: clone_p_vrouter [p_vrouter]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 vip__management        (ocf::fuel:ns_IPaddr2): Started node-2.domain.local
 vip__vrouter_pub       (ocf::fuel:ns_IPaddr2): Started node-2.domain.local
 vip__vrouter   (ocf::fuel:ns_IPaddr2): Started node-2.domain.local
 vip__public    (ocf::fuel:ns_IPaddr2): Started node-3.domain.local
 Clone Set: clone_p_haproxy [p_haproxy]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_p_mysqld [p_mysqld]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Master/Slave Set: master_p_conntrackd [p_conntrackd]
     Masters: [ node-2.domain.local ]
     Slaves: [ node-3.domain.local node-7.domain.local ]
 sysinfo_node-2.domain.local    (ocf::pacemaker:SysInfo):       Started node-2.domain.local
 sysinfo_node-3.domain.local    (ocf::pacemaker:SysInfo):       Started node-3.domain.local
 Master/Slave Set: master_p_rabbitmq-server [p_rabbitmq-server]
     Masters: [ node-3.domain.local ]
     Slaves: [ node-2.domain.local node-7.domain.local ]
 Clone Set: clone_neutron-openvswitch-agent [neutron-openvswitch-agent]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_neutron-l3-agent [neutron-l3-agent]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_neutron-metadata-agent [neutron-metadata-agent]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_neutron-dhcp-agent [neutron-dhcp-agent]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_p_dns [p_dns]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 sysinfo_node-1.domain.local    (ocf::pacemaker:SysInfo):       Stopped
 Clone Set: clone_ping_vip__public [ping_vip__public]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 Clone Set: clone_p_ntp [p_ntp]
     Started: [ node-2.domain.local node-3.domain.local node-7.domain.local ]
 sysinfo_node-7.domain.local    (ocf::pacemaker:SysInfo):       Started node-7.domain.local
root@node-2:~#
```

集群中的所有节点都运行 Ubuntu 16.04（Xenial），该发行版包含了稳定的 Magnum 2.0 软件包，可以直接从上游获取并直接使用 `apt-get install` 安装。

然而，在安装 Magnum 之前，需要准备好环境。首先，需要一个数据库。从任何一个 controller 节点上启动 MySQL 客户端：

```
node-1# mysql
```

在 MySQL 中，创建 magnum 使用的数据库和用户名，并赋予正确的权限：

```
CREATE DATABASE magnum;
GRANT ALL PRIVILEGES ON magnum.* TO 'magnum'@'controller' \
  IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON magnum.* TO 'magnum'@'%' \
  IDENTIFIED BY 'password';
```

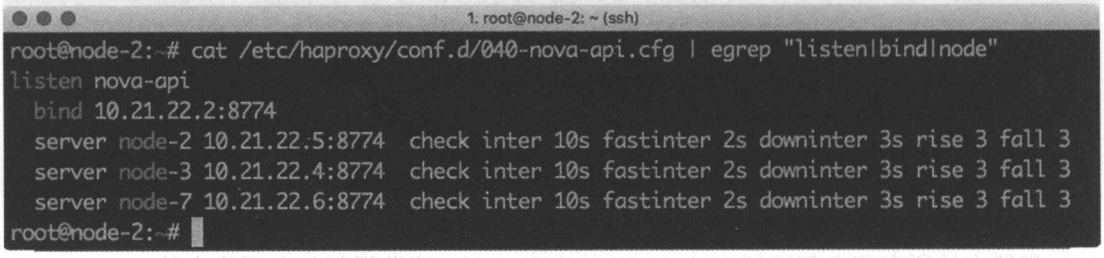
然后，必须在 Keystone 中创建好服务使用的密码。首先定义一个 magnum 的 OpenStack 用户，此用户必须被添加进 services 用户组。services 用户组是一个特别的用户组，它包含 OpenStack 跨集群的一些服务，如 Nova、Neutron 等。

```
openstack user create --domain default --password-prompt magnum
openstack role add --project services --user magnum admin
```

接着，创建一个新的 service：

```
openstack service create --name magnum --description "OpenStack
Container Infrastructure" container-infra
```

OpenStack 的 program 的调用和交流是通过它们的 API 来完成的。一个 API 通过一个端点来访问，这是一对 URL 和端口，它在 HA 的设置中会被 HAproxy 进行代理。在我们的设置中，HAproxy 会在 10.21.22.2 上接受 HTTP 请求，然后在 controller 的 IP 间进行负载，即 10.21.22.4、10.21.22.5 和 10.21.22.6。



我们必须在每个 zone（public，internal 或者 admin）中为 Magnum 创建这些端点，默认监听的端口是 9511：

```
openstack endpoint create --region RegionOne \
  container-infra public http://10.21.22.2:9511/v1
openstack endpoint create --region RegionOne \
```

```
  container-infra internal http://10.21.22.2:9511/v1
openstack endpoint create --region RegionOne \
  container-infra admin http://10.21.22.2:9511/v1
```

同时，magnum 需要额外配置才能在内部以 domain 组织负载，所以还必须添加一个专门的 domain 和一个 domain 用户：

```
openstack domain create --description "Magnum" magnum
openstack user create --domain magnum --password-prompt
magnum_domain_admin
openstack role add --domain magnum --user magnum_domain_admin admin
```

现在，一切都准备就绪，终于可以运行 `apt-get` 进行安装了。在三个 controller 上运行下面的命令，并在弹出来的 ncurses 界面中记得永远回答 No，不要对环境做什么修改，保持默认的配置：

```
apt-get install magnum-api magnum-conductor
```

HA Magnum 安装后的配置

要配置 magnum 是非常简单的，要让它运行起来所需要的做的是：

1. 对 `magnum.conf` 文件进行配置
2. 新启动 magnum 的执行文件
3. 开端口 `tcp/9511`
4. 配置好 HAproxy，使之可以接受 magnum API 请求并进行负载
5. 新加载（reload）HAproxy

在每一个 controller 上必须要做的关键配置如下。首先，在每一个 controller 上，host 参数应该位于管理网络中的接口：

```
[api]
host = 10.21.22.6
```

如果没有安装 Barbican（一个专门用来管理 secret 的一个 OpenStack 项目，如密码），必须使用 `x509keypair` 插件来处理证书：

```
[certificates]
cert_manager_type = x509keypair
```

然后，需要一个数据库连接字符串。在这个 HA 配置中，MySQL 会响应 VIP 10.21.22.2 的请求：

```
[database]
connection=mysql://magnum:password@10.21.22.2/magnum
```

Keystone 的认证的配置如下（其配置项的含义都比较容易理解）：

```
[keystone_authtoken]
auth_uri=http://10.21.22.2:5000/
memcached_servers=10.21.22.4:11211,
10.21.22.5:11211,10.21.22.6:11211
auth_type=password
username=magnum
project_name=services
auth_url=http://10.21.22.2:35357/
password=password
user_domain_id = default
project_domain_id = default
auth_host = 127.0.0.1
auth_protocol = http
admin_user = admin
admin_password =
admin_tenant_name = admin
```

Oslo（消息代理）也必须配置好来负责传递消息：

```
[oslo_messaging_notifications]
driver = messaging
```

RabbitMQ 的配置如下，要指定 Rabbit 的集群主机（因为 Rabbit 运行在 controller 上，需要所有 controller 的管理网络的 IP 地址）：

```
[oslo_messaging_rabbit]
rabbit_hosts=10.21.22.6:5673, 10.21.22.4:5673, 10.21.22.5:5673
rabbit_ha_queues=True
```

```
heartbeat_timeout_threshold=60
heartbeat_rate=2
rabbit_userid=magnum
rabbit_password=A3elbTUIqOcqRihB6XE3MWzN
```

最后，trustee 的附加配置是：

```
[trust]
trustee_domain_name = magnum
trustee_domain_admin_name = magnum_domain_admin
trustee_domain_admin_password = magnum
```

在进行了这些配置更改后，必须要重新启动 magnum 服务：

```
service magnum-api restart
service magnum-conductor restart
```

Magnum 使用的默认端口是 tcp/9511，所以到这个端口的流量都必须在 iptables 里设置成 accepted。修改 iptables 规则，然后添加下面这条规则：

```
-A INPUT -s 10.21.22.0/24 -p tcp -m multiport --dports 9511 -m
comment --comment "117 magnum-api from 10.21.22.0/24" -j ACCEPT
```

其位置就放在其他 OpenStack 服务规则的后边，就在 116 openvswitch db 的后边。

现在到了配置 HAproxy 来接受 magnum 请求的时候了。在 controller 的/etc/haproxy/conf.d 目录下面添加 180-magnum.cfg 文件，内容如下：

```
listen magnum-api
  bind 10.21.22.2:9511
  http-request  set-header X-Forwarded-Proto https if { ssl_fc }
  option  httpchk
  option  httplog
  option  httpclose
  option  http-buffer-request
  timeout  server 600s
  timeout  http-request 10s
  server node-1 10.21.22.6:9511  check inter 10s fastinter 2s
  downinter 3s rise 3 fall 3
  server node-2 10.21.22.5:9511  check inter 10s fastinter 2s
```

```
    downinter 3s rise 3 fall 3
    server node-3 10.21.22.4:9511  check inter 10s fastinter 2s
    downinter 3s rise 3 fall 3
```

这里配置的 magnum-api 在 VIP `10.21.22.2:9511` 上监听，负责代理后面三个 controller 节点。

然后，必须在 Pacemaker 中重启 HAproxy。在任意一个 controller 中，运行：

```
pcs resource disable p_haproxy
```

然后等所有 controller 上的 HAproxy 进程都启动起来（可以用 `ps aux` 来检查），这个过程很快，应该要不了 1 秒，接着执行：

```
pcs resource enable p_haproxy
```

然后，magnum 服务应该就启动起来并且可用了：

```
source openrc
magnum service-list
```

在 Magnum 上创建一个 Swarm 集群

创建一个 Swarm Cluster，在 magnum 添加这个容器编排引擎的时候，需要以下步骤：

1. 新建一个 Swarm Mode 模板
2. 从模板启动一个集群

对于尚未存在的事情我们这里不会深入探讨，但是我们可以想象一下这个命令大概会是这样的：

```
magnum cluster-template-create \
--name swarm-mode-cluster-template \
--image-id ubuntu_xenial \
```

```
--keypair-id fuel \
--fixed-network private \
--external-network-id public \
--dns-nameserver 8.8.8.8 \
--flavor-id m1.medium \
--docker-volume-size 5 \
--coe swarm-mode
```

在这里，我们定义了一个类型为 swarm-mode 的集群模板，它基于 Ubuntu Xenial，然后使用 `m1.medium` 配置：VM 会被注入 fuel 的秘钥对，然后会有一个外部的公网 IP 地址。基于这样一个模板创建一个集群的用户体验大概像下面这样：

```
magnum cluster-create --name swarm-mode-cluster \
    --cluster-template swarm-mode-cluster-template \
    --manager-count 3 \
    --node-count 8
```

在这里，启动了一个包含 3 个管理器，5 个 worker 的集群。

magum 是一个伟大的项目，它位于用来在 OpenStack 上运行容器编排的抽象层的最高点。它在 Rackspace cloud 上运行，并且可以通过 Carina 来公开使用，相关内容请参考：http://blog.rackspace.com/carina-by-rackspace-simplifies-containers-with-easy-to-use-instant-on-native-container-environment。

本章小结

在本章中，我们探索了其他可以运行 Docker Swarm 的平台。我们使用了最新的 Docker 工具，即 Docker for AWS 和 Docker for Azure ，然后我们使用它们演示了如何用新的方法来安装 Swarm。在介绍了 Docker Datacenter 后，我们探索了私有云的部分。我们使用了 OpenStack，展示如何在其上面运行 Docker，如何安装 OpenStack Magnum，以及如何在它上面创建 Swarm 对象。我们即将完成我们的 Swarm 旅途了。

下一章我们将会展望 Docker 编排未来发展的蓝图。

第 11 章

Swarm 的未来展望

Docker 的生态正在形成一幅宏大的画卷，而 Swarm 将会是其中一个核心组件。让我们来设想一张路线图并展望 Swarm 的未来。

Provisioning 的挑战

对于大规模 Swarm 集群的创建，现在还没有官方工具。在目前，正如我们已经在前面的章节中见到的，运维人员需要使用自己编写的脚本，或者使用一些临时工具（例如 Belt）、配置管理器（如 Puppet 或者 Ansible）或编排模板。最近又多了两个选择，即 Docker for AWS 和 Docker for Azure。

但是这个情况很可能有望被软件定义为基础设施（Software defined infrastructure）工具集，以一种统一的方式解决掉。

软件定义基础设施

从开始把容器作为构建单元（building block），然后在构建系统对应用和基础设施进行架构、编排、伸缩、安全加固并且部署，可编程的互联网（programmable Internet）可能是一个长期的目标。

继 SwarmKit（用来编排的工具包）之后，Docker 在 2016 年 10 月开源了 Infrakit，这是一个基础设施的工具包。

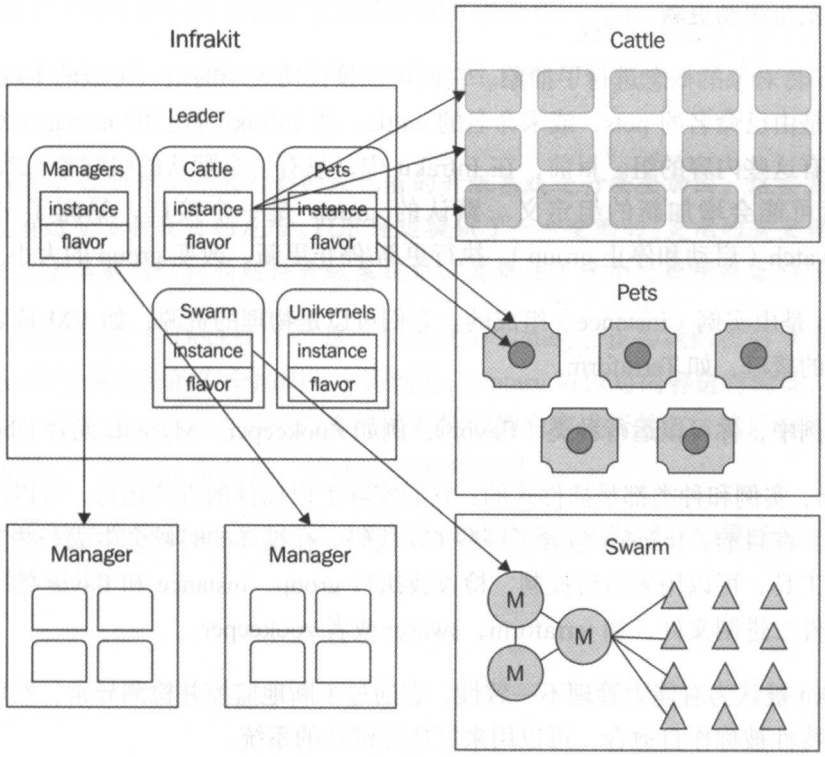
Infrakit

Docker Engine 重心在容器，Docker Swarm 重心在编排，而 Infrakit 重心在把 group（组）作为一种原语。group 可以是任何的对象：Pets、cattle、unikernel 或 Swarm 集群。

Infrakit 能解决在不同基础设施上管理 Docker 的问题。在没有 Infrakit 的时候，这个问题十分困难，而且解决方案不具有移植性。它的理念是从对数据中心进行架构到运行简单的容器，提供一致的用户体验。Infrakit 是目前 Docker 公司对于创建可编程基础设施最高层的抽象，它是这么进行自我描述的：

> “Infrakit 是一个用来创建并管理声明式的、自愈合的基础设施的工具包。它将基础设施的自动化分解成了简单的可插拔的组件，这些组件可以一起来积极地保证基础设施的状态符合用户的规范要求。”

Infrakit 在整个构架中靠在容器引擎的一边：



这个组织形式是按照 group 来的。有一个 group 来负责 Infrakit 自己的结构，由用来保持配置的 manager 组成。在同一个时间，只有一个管理者和一定数量的追随者，比如两个。每一个 manager 都包含一些组的声明。group 可能是 cattle、pets、swarm 或 unikernel 等。每一个 group 通过实例（instance，真实的资源，如容器）和种类（flavor，资源的类型，如 Ubuntu Xenial 或者一个 MySQL Docker 镜像）来定义。

Infrakit 是声明式的。它依赖 JSON 配置文件然后在内部使用了大家熟悉的封装和组合模式，让配置作为输入，然后对配置进行处理，并让基础设施汇聚（converge）成一个特定的配置状态。

Infrakit 的目标是：

- 提供一个统一的工具集来管理 group
- 可插拔
- 提供自愈合
- 发布滚动更新

group 将对象的概念进行了抽象。它们可以是任何大小的组，可以向上或者向下伸缩，它们可以是由已命名的 pets，或未命名的 cattle，或 Infrakit 自己的 manager 组成的组，或者包含所有这些内容的组。目前，在 Infrakit 中，只有一个默认的组配置（默认的插件），但是稍后可能会增加新的组定义。默认的 group 是一个接口，暴露了一些操作如：watch/unwatch（启动和停止 group），执行更新/停止更新，改变 group 的大小。

group 是由实例（instance）组成的。它们可以是物理的资源，如 VM 或者容器，或者其他服务的接口，如 Terraform。

在实例中，你可以运行种类（flavor），例如 Zookeeper、MySQL 或者 Ubuntu Xenial。

group、实例和种类都是插件式的：它们实际上以插件的方式运行，可以由任意的编程语言编写。在目前，Infrakit 包含了一些 Go 代码，在编译的时候会生成一些二进制文件，如命令行工具，可以用来执行控制、检查或执行 group、instance 和 flavor 的一些操作，还有一些插件二进制文件，如 terraform，swarm 或者 zookeeper。

Infrakit 被认为有能力管理不一致性，它通过不断地监控并检测异常，然后触发响应动作。这种特性被称作自愈合，可以用来打造更健壮的系统。

Infrakit 支持的一个重要操作是可以发布滚动更新来对 instance 进行升级。如升级一个容器中的软件包，更新一个容器的镜像，或者其他什么。这是通过使用 TUF（The Udpate Framework，升级框架）来完成的，这个项目会在下一节讲到。

Infrakit 在本书写作的时候，还比较年轻，因此我们除了 Hello World 级别的例子，没有其他什么好拿来展示。在网上很快就会有 Infrakit 的各种 Hello World 例子，然后 Infrakt 团队会发布一个 step-by-step 的教程，以及使用其的 file 或 Terraform 插件。我们可以把它描述成 Docker 生态系统中的架构层，并期待它有望用来部署好 Swarm、provision 主机，然后设置好它们之间的连接。

Infrakit 有望被包含在引擎中，并在 1.14 版本中作为一个实验性特性。

TUF—— The Update Framework

在 2016 年柏林 Docker 峰会上，讨论了另外一个主题——TUF（https://theupdateframework.github.io/），这个工具集的目标是提供安全的滚动升级方式。

已经有了很多在实际中用来升级的升级工具，但 TUF 却要更胜一筹。该项目的首页是这么说的：

> “TUF 会帮助开发者来对新的或已有的升级系统进行安全加固，这些系统通常被发现有遭受多种攻击的危险。TUF 通过提供了一个全面的、灵活的安全框架，并能让开发者集成到任何的软件升级系统中，其解决了这个大规模存在的问题。”

TUF 已经被整合进了 Docker 中，这个工具就是 Notary。正如我们在第 9 章“Swarm 集群和 Docker 软件供应链的安全加固”中讲到的，Notary 可以对内容进行验证，让系统轻松地管理秘钥。有了它，开发人员可以离线地对他们的内容进行签名，然后将已签名的受信任集合推送到一个 Notary 服务器，并让这些内容可用。

TUF 是否会被整合进 Docker Infrakit 并作为滚动升级的机制呢？如果是的话，这将会是一个了不起的进步。

Docker Stacks 和 Compose

另外一个开发者已经可以使用，但是目前还处于试验阶段的特性是 Stacks。我们在第 6 章“Swarm 上真实应用的部署”中介绍了 Stacks。它们将会是在 Swarm 上部署应用的默认方式。它的想法是把打包成捆（bundle）的一组容器一并启动，而不是动态式地加入容器。

同时，我们也可以期待 Compose 和新的 Swarm 将会整合到一起。

Caas ——容器即服务

在 XaaS 的时代，一切都可以被看做是一个软件，不仅仅容器成为了头等公民，编排系统和基础设施也即将成为头等公民。所有的这些抽象将会带来运行该工具生态的一种云定义方式：容器即服务。

一个 CaaS 的例子是 Docker Datacenter。

Unikernel

我们说过，Swarmkit 作为一个工具包，不仅仅能够运行容器的集群，同时也能运行 unikernel 的集群。

那 unikernel 是什么东西，它强大之处在哪里呢？

如果你使用过 Docker For Mac，那你已经在使用 unikernel 了。它是这些系统的核心。在 Mac 上，xhyve——一个 FreeBSD 虚拟系统（bhyve）的一个移植版，会把一个 Docker 主机以 unikernel 的模式运行。

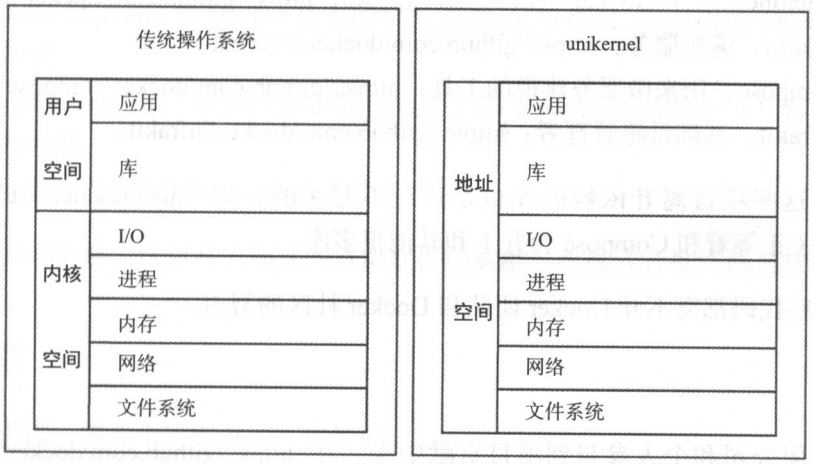
人人都爱容器，因为它们体积小并且速度快。但是使用一种对内核进行抽象的机制，并让它的组件共享系统的资源、库和执行文件，这里的安全后患不能忽视。可以在任一一个搜索引擎上检索一下容器相关的 CVE 列表。你会发现，这个后患很严重。

Unikernel 承诺要在最高的层次来对软件的架构进行重新评估。这里做一个简单的解释。它有一个很有效的方式来保证安全性最大化，并且有自身运行的体积非常小的特点。在一个我们频繁使用 TB、PB 以及更大单位来描述数据大小的世界里，当你听说一个 KVM 的 unikernel 实现，如 ukvm 大小不超过 67KB、一个 Web 服务器二进制大小不超过 300KB 或

者一个操作系统镜像仅有几兆大小的时候，你有什么想法？

这是因为它们不会将所有的系统调用暴露给架构，但是这些调用被包含在了二进制自身里面。一个 ping 二进制文件不需要任何访问磁盘的系统调用，也不需要使用加密方面的函数或者管理系统进程。那么为什么不把这些调用从 ping 中拿走，然后只提供它需要的最少的东西呢？这就是 unikernel 背后的理念。一个 ping 命令可能只需要和一些网络 I/O 以及原始套接字文件进行编译而不需要其他。

在 Unikernel 中，没有内核空间和用户空间区别，因为地址空间被统一了。这意味着，地址表是连续（continuous）的。正如之前解释的，这是因为 unikernel 的二进制文件编译的时候，其在二进制文件中嵌入了它们需要的系统函数，如 I/O 操作、内存管理或者共享库。在传统的操作系统模型中，应用会在运行时查找并使用系统调用，然而在 unikernel 中，这些系统调用在编译的时候被静态链接了。



第一眼看去，可能很奇怪，但这是一个进程隔离和安全方面的重大进步。即使有人闯进了运行着 unikernel 的系统，他也几乎找不到什么安全漏洞。攻击面是如此窄，除了个别被用到的肯定已经硬编码的操作系统调用或者特性，很难发掘一些没有使用的系统调用或者特性。没有可以启用的 shell，没有外部的工具或者脚本，没有配置文件或者密码文件，也没有额外绑定的端口。

那 unikernel 和 Docker 之间会发生什么事情？

在 2015 年巴塞罗那的 DockerCon EU 上，有公司在台上展示了如何整合 Docker 和 unikernel，然后 Docker 公司最终收购了该公司，这预示了一些事情包括 Docker for Mac 的诞生。

在柏林 Docker 峰会上，有一些人提到了在 SwarmKit 中让 unikernal 和容器一起运行。整合的未来即将来临。

为 Docker 做贡献

所有 Docker 中的革新之所以成为可能都得益于这些项目依赖广大的社区。Docker 是一个非常复杂和积极的项目，它分成了好几个 Github 仓库，最有名的是：

- Docker 自己，引擎部分：https://github.com/docker/docker。
- Machine，一个 Docker 主机的实例化工具，https://github.com/docker/machine。
- Swarm，编排服务：https://github.com/docker/swarmkit。
- Compose，用来微服务建模的工具：https://github.com/docker/compose。
- Infrakit，基础设施管理者：https://github.com/docker/infrakit。

同时，这些项目离开依赖的库也是运行不起来的，如 Libcontainer、Libnetwork、Libcompose（正等着和 Compose 合并）和其他很多库。

所有这些代码都离不开 Docker 团队和 Docker 社区的努力。

Github

欢迎任何公司和个人参与到项目贡献中来，在 https://github.com/docker/docker/blob/master/CONTRIBUTING.md 可以找到一些指导性的内容。

提交 issue

报告一些在使用过程中遇到的异常和 Bug，或者在 Github 的相关项目中提交 issue 发表想法，是一个让自己开始参与项目的很好的方式。

代码

提交 pull request（简称 PR）来修复问题或者提议新特性也是一种非常受欢迎的方式。这些 PR 应该引用 issue 列表中的 issue，并相应遵守指南中的要求。

belt 和其他项目

同时，在本书写作的时候，有一些项目启动了：

- Swarm2k 和 Swarm3k，这是一个以社区为中心的用来创建大规模的 Swarm 的实验。Swarm 的代码、教程和结构都可以在 https://github.com/swarmzilla 相应的仓库中找到。
- belt 是一个 Docker 主机的 Provisioner。在目前，它只包含一个 DigitalOcean 的驱动，但是可以被扩展。
- Swarm、Machine 和 Docker 证书的相关的 Ansible 模块，可以用在 Ansible playbook 中找到。
- Docker Hub 中的一些用来展示特定组件（如 `fsoppelsa/etcd`）或者介绍新特性的容器。
- 其他较小的一些 PR，hack 或者代码。

遵守开源的精神，任何对上面这些自由软件的贡献、改进或者批评都是欢迎的。

本章小结

最后，讲讲一些关于这本书的一些历史：

当写作一本关于 Docker Swarm 书籍的项目的时候，当时还只有 Docker Swarm 的单机模式，当时一个 Swarm 容器负责容器基础设施的编排，还必须依赖外部的发现系统，如 Etcd、Consul 或者 Zookeeper。

仅仅几个月，再回首这段时间，就像在回顾远古历史一样。在 2016 年 6 月底，当编排工具集 SwarmKit 被开源，作为 Swarm Mode 包含在了引擎中，Docker 在编排方面迈出了巨大的一步。它发布了一个编排 Docker 完整的、可伸缩的和默认安全的方式。然后，人们发现编排 Docker 最好的方式是 Docker 自己。

但是随着 2016 年 10 月 Infrakit 被开源，Docker 在基础设施方面又迈出了一大步。现在不仅编排和容器组是原语，其他对象（包括容器、虚拟机、unikernel 甚至实体机）的组，即使混合在原始的 Infrakit Intent 中，也是原语。

在很快的将来，我们有望看到所有这些项目整合在一起，Infrakit 作为基础设施管理者负责任何 Swarm 的 provision，进行容器和其他对象的编排、连接、存储（完全有状态）、滚动升级、被 overlay 网络连接和安全加固。

Swarm 只是这个宏大的生态系统的一个开始。

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Swarm容器编排与Docker原生集群

Docker Swarm 是 Docker 生态体系中最关键的组件之一，它提供了原生的容器编排解决方案。随着最新的改进，它正在成为构建 Docker 集群的一个首选。

本书涵盖了 Swarm、Swarm Mode 和 SwarmKit。它循序渐进地引导读者探索 Swarm 的运作原理并掌握 Swarm 的使用技巧。全书从讲解如何搭建一个本地测试环境开始，然后将主题转移到大型分布式基础设施，并接着讨论了 Swarm 内部是如何运作的，SwarmKit 有哪些新特性，如何将大规模的 Swarm 部署自动化，以及如何在公有云和私有云中配置并运营一个 Swarm 集群。

书中随后将焦点转到大型生产级应用上，并在 Swarm 中部署大规模容器。它同时包含了众多高级主题，如卷、调度、Libnetwork 的深入探索、安全和平台的伸缩性。

在本书的结尾，你将会理解 Swarm 是如何精妙地与 Docker 生态体系整合的，你无须重构容器应用便可适配其他的平台。

你将会学到：

- 创建并管理任意大小的 Swarm Mode 集群
- 深入了解创建迄今为止最大的 Swarm 集群的幕后，即 Swarm 2k 和 Swarm 3k，分别有 2300 和 4700 个节点
- 理解发现机制和 Raft
- 在 Swarm 上部署容器化应用
- 管理 AWS、Azure 和 DigitalOcean 上的 Swarm 集群
- 在 Swarm 中整合 Flocker 卷
- 在 OpenStack Magnum 上创建并管理 Swarm

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：徐津平
封面设计：吴海燕

上架建议：架构/开发/运维

ISBN 978-7-121-31792-7

定价：69.00元